공학박사학위논문

대규모 분산 시스템을 위한 효율적인 분산 트리거 계수 알고리즘

Efficient Distributed Trigger Counting Algorithms for Large-scale Distributed Systems

2013년 8월

서울대학교 대학원
전기 · 컴퓨터 공학부
김 석 현

공학박사학위논문

대규모 분산 시스템을 위한 효율적인 분산 트리거 계수 알고리즘

Efficient Distributed Trigger Counting Algorithms for Large-scale Distributed Systems

2013년 8월

서울대학교 대학원
전기 · 컴퓨터 공학부
김 석 현

대규모 분산 시스템을 위한 효율적인 분산 트리거 계수 알고리즘

Efficient Distributed Trigger Counting Algorithms for Large-scale Distributed Systems

지도교수 조유근

이 논문을 공학박사 학위논문으로 제출함
2013년 5월

서울대학교 대학원
전기 · 컴퓨터 공학부
김 석 현

김석현의 공학박사 학위논문을 인준함
2013년 6월

위원장	신 현 식	(인)
부위원장	조 유 근	(인)
위 원	박 근 수	(인)
위 원	민 상 렬	(인)
위 원	박 용 수	(인)

초 록

시스템에 참여한 노드의 수가 n이고 검출해야 하는 트리거의 수가 w라 하자. 분산 트리거 계수 알고리즘은 n개 노드가 검출한 전체 트리거의 수가 w가 될 때 이를 사용자에게 알려준다. 트리거의 분포에 대한 어떤 통계적 정보도 미리 주어지지 않으며 트리거의 수는 노드의 수에 비해 매우 많다고 가정한다.

분산 트리거 계수 알고리즘은 분산 모니터링 및 전역 스냅샷 알고리즘에 응용 가능하다. 분산 모니터링은 다양한 분산 시스템에서 시스템의 내부 및 외부를 감시하는데 사용된다. 전역 스냅샷 알고리즘은 분산 시스템 전체의 상태를 저장하여 시스템 복구를 위한 체크 포인트를 (check point) 만드는 용도로 사용된다.

본 논문에서는 대규모 분산 시스템을 위한 효율적인 분산 트리거 계수 알고리즘인 *TreeFill*과 *TreeFill-p*를 제시한다. 이들 알고리즘은 라운드를 (rounds) 기반으로 트리거를 검출한다. 각 라운드에서 전체 트리거 중 일부를 검출하고 검출된 트리거의 수가 w가 되면 사용자에게 이를 알려준다. *TreeFill*은 분산 트리거 계수를 위해 높은 확률로 $O(n\log(w/n)$개의 메시지를 사용한다. 이는 분산 트리거 계수를 위한 정확한 알고리즘들의 (exact algorithms) 메시지 하한을 만족한다. 또한 *TreeFill* 알고리즘이 구동할 때 각 메시지가 수신하는 최대 메시지의 수는 높은 확률로 $O(\log(w/n))$이 된다. *TreeFill-p*는 확률적 알고리즘으로써 낮은 실패 확률을 갖지만, $w = O(n^m)$인 경우 $(m > 0)$ 분산 트리거 계수를 위해 높은 확률로 $O(n)$개의 메시지를 사용한다. 또한 *TreeFill-p*에서 각 노드가 수신하는 메시지의 수는 높은 확률로 $O(1)$이다. 본 논문에서는 *TreeFill* 및 *TreeFill-p* 알고리즘의 성능을 증명하고 다양한 시뮬레이션을 통해 이들 알고리즘의 성능을 검토 하였다.

주요어: 분산 트리거 계수 알고리즘, 분산 시스템, 분산 알고리즘, 분산 모니터링, 전역 스냅샷

학번: 2008-30213

목 차

표 목 차

그 림 목 차

제 1 장 서론

분산 트리거 계수 알고리즘은 (distributed trigger counting algorithm) 대규모 분산 시스템에서 시스템 모니터링과 전역 스냅샷을 (global snapshot) 위해 사용될 수 있다. 시스템 모니터링은 여러 분산 시스템에서 시스템의 상태 분석 및 유지 관리를 위해 유용하게 사용되는 알고리즘이다. 시스템 모니터링 기능을 통하여 시스템의 현재 상태를 파악하고 이를 통해 전체 시스템을 보다 효율적으로 관리할 수 있다. 분산 트리거 계수 알고리즘은 시스템 모니터링 기능에 유용하게 사용 가능하다.

전역 스냅샷은 여러 노드들로 (또는 프로세서들) 구성된 분산 시스템에서, 전체 시스템을 구성하는 각 노드와, 노드 간의 통신 채널의 상태를 일관성 있게 (consistent) 저장하는 것을 말한다. 저장된 전역 스냅샷은 이후 시스템 복구 및 기타 용도를 위해 사용 가능하다. 전역 스냅샷을 시스템 복구 용도로 사용할 경우, 일반적으로 여러 전역 스냅샷들을 주기적으로 만들게 된다. 시스템에 치명적인 문제가 발생한 경우, 가장 최근에 저장된 전역 스냅샷의 상태로 전체 시스템을 되돌림으로써 시스템 문제로 인한 피해를 최소화 할 수 있다. 분산 트리거 계수 알고리즘은 대규모 분산 시스템을 위한 전역 스냅샷 알고리즘의 주요 부분으로써 사용 가능하다. 효율적인 분산 트리거 계수 알고리즘의 설계는 효율적인 전역 스냅샷 알고리즘의 설계를 위해 중요하다.

본 논문은 대규모 분산 시스템을 위한 효율적인 두 가지 분산 트리거 계수 알고리즘, *TreeFill*과 *TreeFill-p*를 제안한다. 본 장에서는 먼저 1.1절에서 분산 트리거 계수 문제를 정의하고 분산 트리거 계수 알고리즘들을 비교하기 위한 성능 지표를

제시한다. 다음으로 연구 배경으로써 1.2절에서 분산 트리거 계수 알고리즘과 분산 모니터링의 관계를, 1.3절에서 분산 트리거 계수 알고리즘과 전역 스냅샷 알고리즘의 관련성을 살펴본다. 연구 목적과 범위를 1.4절에서 규정하고 1.5절에서 연구 성과를 요약하여 기존 분산 트리거 계수 알고리즘들과 제안 알고리즘들의 성능 비교를 제시한다. 마지막으로 1.6절에서 논문의 구성을 언급한다.

1.1 분산 트리거 계수 문제

분산 트리거 계수 문제는 (distributed trigger counting problem) n개의 노드로 구성된 분산 시스템에서 w개의 트리거들을 (triggers) 감지하는 문제이다. 트리거들의 통계적 분포나 수학적 모델은 미리 주어지지 않는다. 트리거들은 임의의 시간에 임의의 노드에서 감지된다. 이러한 트리거들이 n개 노드에서 감지되는 동안, 전체 노드에서 감지된 트리거의 총 수가 w가 되는 시점에 사용자에게 이를 알려야 한다. 통상 분산 트리거 계수 문제에서 $w \gg n$이라고 가정한다. $w \leq n$인 경우 각 노드가 트리거를 감지할 때 마다 관리 노드 (coordinating node)에게 메시지를 보내면 관리 노드에서는 $O(n)$ 개의 메시지를 이용하여 트리거를 검출할 수 있다. 그러나 $w \gg n$인 경우는 이렇게 쉽게 해결하기 어렵다.

분산 트리거 계수 알고리즘에서 트리거는 다소 추상적인 개념으로써, 시스템에서 발생하는 어떤 이벤트들이나, 사용자가 관심을 가지고 있는 시스템 내부 상태 변화를 트리거에 대응시킬 수 있다. 예를 들어 데이터 센터에서 특정 서비스를 이용하고 있는 사용자의 수를 추적하는 경우를 생각해 보자. 1000대의 서버에서 관찰 대상인 응용이 동작하고 있다고 하자. 하나의 서버에서 서비스 가능한 사용자 수는 3000 명이라 하자. 그러면 동시에 서비스 가능한 인원은 3백만 명이다. 사용자 수의 증감을 추적하기 위해 최대 서비스 이용 가능 인원의 1%인 삼만 명을 w로 설정하자. 사용자 한 명이 해당 응용을 새로 이용하기 시작하는 경우 a 타입 트리거를 발생하고, 이용을 종료하면 b 타입 트리거를 발생한다. 전체 1000대의 서버는 트리거 a, b에 대하여 각각 분산 트리거 계수 알고리즘을 구동한다. 그러면 현재 해당 응용을 사용하고 있는 사용자의 수를 삼만명 단위로 알 수 있다. 만약 일주일 동안 a타입 트리거가 152만 개 발생하고 b타입 트리거가 120만 개 발생했다면 a, b 트리거 계수 과정에서 시스템이 사용자에게 보낸 알람의 수는 각각 $50, 40$이다. 따라서 이를

통해 추정한 관찰 대상 응용을 사용하고 있는 사용자의 수는 $(50-40)$에 삼만을 곱하여 삼십만 명이 된다. 실제로 접속한 사람은 삼십이만 명이므로 이만 명의 오차가 있는데 이는 w 단위로 사용자가 이용자 수의 증감을 알 수 있기 때문이다. 분산 트리거 계수를 사용할 때 w를 크게 하면 모니터링의 정확도는 떨어지지만 더욱 적은 오버헤드를 이용하여 모니터링이 가능하고, w를 작게 하면 모니터링의 정확도는 증가하나 모니터링 오버헤드 역시 증가한다.

기존 연구들을 살펴보면 분산 트리거 계수 알고리즘의 성능을 비교하기 위하여 주로 다음의 두 가지 성능 지표를 이용한다 [GGS10, CCGS11, CCS11].

- 메시지 복잡도 (Message complexity): w개의 트리거를 검출하기 위해 n개의 노드들이 주고 받은 메시지 수의 총 합.
- 각 노드가 수신한 메시지 수의 최대 값 (MaxRcv): w개의 트리거를 검출하는 과정에서 가장 많은 메시지를 수신한 노드가 받은 메시지의 수.

메시지 복잡도는 분산 트리거 계수 알고리즘이 전체 분산 시스템에 부과하는 네트워크 사용량 측면에서의 오버헤드와 관련이 있다. 전체 노드가 분산 트리거 계수를 위해 송수신한 메시지의 총 양은 (전체 메시지 수 $\times$ 메시지 크기)로 나타낼 수 있다. 기존의 분산 트리거 계수 알고리즘들과 본 논문에서 제시하는 두 가지 알고리즘은 모두 $O(1)$ 크기의 메시지를 사용한다 [GGS10, CCGS11, CCS11]. 따라서 본 논문에서 앞으로 메시지 복잡도를 말할 때는 알고리즘에서 사용한 메시지의 총 수를 말하며, 메시지 크기가 $O(1)$이므로 사용한 메시지의 총 수는 실제로 알고리즘을 구현했을 때 필요한 네트워크 사용량에 비례한다.

분산 트리거 계수 알고리즘에서 각 노드는 분산 트리거 검출을 위해 일정 수의 메시지를 수신하게 된다. MaxRcv는 이 과정에서 가장 많은 수의 메시지를 수신한

노드가 받은 메시지의 수 이다. MaxRcv는 분산 트리거 계수 알고리즘의 확장성과 (scalability) 관련이 있다. 만약 어떤 분산 트리거 계수 알고리즘의 MaxRcv가 $O(n)$ 이라면 시스템의 크기가 커짐에 따라 언젠가 노드가 사용 가능한 네트워크 대역폭의 한계에 다다를 것이고, 전체 시스템에 참여 가능한 노드의 수는 메시지가 집중되는 특정 노드가 사용한 네트워크 대역폭에 제한될 것이다. 만약 MaxRcv가 낮다면 더 큰 시스템에서 원활하게 분산 트리거 계수 알고리즘을 구동할 수 있다.

앞으로 본 논문에서 분산 트리거 계수 알고리즘의 성능을 비교 분석할 때는 '메시지 복잡도', 'MaxRcv' 두 가지 기준을 사용하기로 한다.

1.2 분산 트리거 계수 알고리즘과 분산 모니터링

분산 모니터링 기술은 여러 분산 시스템에 널리 사용되는 핵심적인 기술 중 하나이다. 무선 센서 네트워크는 특정 지역에 배포된 센서들을 이용하여 주변 환경의 다양한 물리적 특성을 감지하고 이를 사용자에게 전달해 준다 [ASSC02, AK04]. 또한 그리드 컴퓨팅, 클러스터 컴퓨팅, 무선 센서 네트워크, 피어-투-피어 컴퓨팅 등의 다양한 종류의 분산 시스템에서 시스템 내부 상태를 관찰하기 위하여 분산 모니터링 기술이 사용된다 [MCC04, ZC04, PP06, CDR08, LC10]. 그리드 및 클러스트 컴퓨팅에서는 시스템의 전역적인 현재 상태를 추정하기 위해서 전체 시스템 상태를 모니터링 하는 소프트웨어들을 사용한다. 이러한 시스템 상태 모니터링 소프트웨어에는 Ganglia [MCC04], GridICE [ADBF+05] 등이 있다.

일반적으로 분산 모니터링을 위해 사용되는 알고리즘에는 중앙 집중 방식, 가십 기반 방식 (gossip-based aggregation), 트리 기반 방식이 (tree-based aggregation) 있다 [MCC04, CDR08, CMY11, JBA11]. 중앙 집중 방식은 중앙의 관리 노드에서 (coordinating node) 다른 모든 노드들이 분석한 정보를 모아서 시스템 전체의 정

보를 얻는 방식이다. 중앙 집중 방식에서는 명백하게 중앙의 관리 노드에 모니터링 오버헤드가 집중되므로 시스템 전체의 확장성 측면에서 단점을 지닌다. 그러나 중앙 집중 방식의 모델은 다양한 모니터링 함수들을 제공한다. 예를 들어 합계, 평균, 최대, 최소 같은 기본적인 기능은 물론이며 엔트로피, 모멘텀 등의 복잡한 함수들을 효율적으로 계산하는 알고리즘이 제안 되었다 [CMY11]. 가십 기반 방식은 분산 시스템에 참여하는 노드들이 자주 참여 및 이탈을 하는 경우 유리하다. 가십 기반 방식에서는 중앙 집중 방식이나 트리 기반 방식과 다르게 정형화된 노드들의 네트워크를 구성하지 않는다. 따라서 주기적으로 가십 알고리즘을 구동하는 것으로 모니터링 하고자 하는 시스템 특성의 평균을 얻을 수 있다 [JBA11].

가장 일반적으로 사용되는 방식은 트리 기반 방식이라 할 수 있다. 센서 네트워크, 그리드 시스템 등에서 트리 기반 방식으로 전체 시스템을 모니터링 하는 기법들이 사용된다 [MCC04, CDR08]. 만약 노드들 사이에 트리 네트워크를 구축할 수 있다면 트리 기반 방식은 $O(n)$개의 메시지를 사용하여 효율적으로 전체 시스템을 모니터링 할 수 있다. 통상 트리 기반 방식은 주기적으로 작동한다. 주기적으로 노드들의 트리 네트워크를 통해 전체 시스템의 정보를 수집함으로써 전체 분산 시스템의 정보를 모니터링 한다.

분산 트리거 계수를 활용한 모니터링 방식은 트리 네트워크를 활용한 주기적인 방식과 관련과 비교하여 다른 특성을 가지고 있다. 두 가지 모니터링 방식을 비교하기 위하여 데이터 센터에 접속 중인 사용자의 수를 분산 집계하는 경우를 생각해 보자. 트리 네트워크를 이용한 주기적인 모니터링 방식은 모니터링 주기 p에 한번씩 전체 시스템에 접속한 사용자의 수를 체크한다. 분산 트리거 계수 알고리즘을 사용하면 전체 시스템에 접속한 사용자의 수가 w만큼 증가하거나 감소하는 시점을 알 수 있다. 그림 1.1은 동일한 사용자 수 변화를 주기적 모니터링과 분산 트리거 계수를 이용한 모니터링을 이용하여 관찰할 때의 차이점을 보여준다.

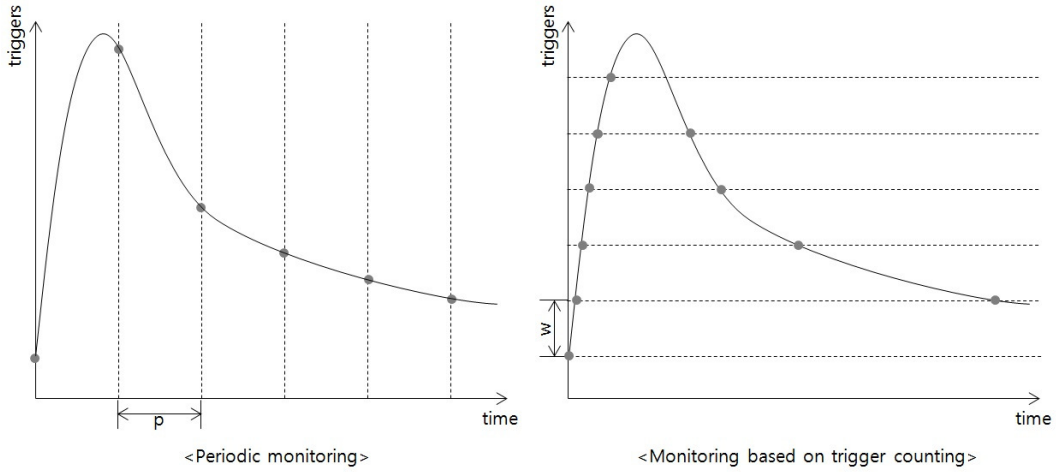

그림 1.1: 주기적 모니터링과 트리거 계수 기반 모니터링의 차이.

만약 데이터 센터에 접속 중인 사용자 수가 급격히 변화할 경우 주기적 모니터링은 모니터링 주기인 p보다 빨리 이러한 변화를 감지할 수 없다. 하지만 분산 트리거 계수 알고리즘을 사용하면 미리 정해놓은 w개의 트리거가 발생하는 시점을 알 수 있으므로 전체 시스템의 급속한 변화에 보다 빠르게 대응할 수 있다. 반대로 데이터 센터에 접속 중인 사용자의 수에 크게 변화가 없을 경우 분산 트리거 계수 알고리즘은 접속자 수의 증감에 비례하여 모니터링이 이루어지므로 적은 수의 메시지만을 사용하여 모니터링 작업을 수행할 수 있다. 하지만 주기적 모니터링 방식은 이러한 모니터링 대상의 특성과 관계없이 모니터링 오버헤드가 항상 모니터링 주기 p에 따라 결정된다.

아직 분산 트리거 계수 알고리즘을 실제 분산 모니터링에 활용한 실제 시스템 구현은 발표되지 않았다. 분산 모니터링 응용에 분산 트리거 계수 알고리즘의 사용 가능성에 대한 언급만을 찾아 볼 수 있다 [CCS11, KLPC13]. 하지만 분산 트리거 계수 알고리즘은 모니터링 대상의 변화에 신속하게 반응할 수 있으므로 대규모 분산 시스템에서 전체 시스템의 변화에 빠르게 대응하는 응용이 필요한 경우 분산

트리거 계수 알고리즘이 유용하게 활용될 수 있는 잠재력이 있다.

1.3 분산 트리거 계수 알고리즘과 전역 스냅샷

현재 분산 트리거 계수 알고리즘이 직접적으로 응용 가능한 분야는 전역 스냅샷 (global snapshot) 알고리즘이다 [GGS10, CCGS11, CCS11]. 분산 시스템에서 전역 스냅샷은 주기적으로 시스템 전체의 상태를 저장하는 용도로 활용된다. 시스템에 문제가 발생한 경우 가장 최근의 스냅샷에 저장된 상태로 시스템 전체의 상태를 복구함으로써 시스템의 문제로 인한 피해를 최소화 할 수 있다.

대규모 분산 시스템을 고려하지 않은, 기존의 전역 스냅샷 알고리즘을 대규모 분산 시스템에 적용할 경우 노드 간의 통신 비용이 크게 증가한다 [GGS10, Ksh10, Tsa13]. 이러한 통신 비용 증가의 주된 이유는 전역 스냅샷을 저장할 때, 노드 간의 채널 상태들을 (channel states) 저장하는 비용이 전체 노드의 수가 n이라 할 때 일반적으로 $O(n^2)$이기 때문이다 [CL85, LY87, Mat93]. 분산 트리거 계수 알고리즘을 활용하면 전역 스냅샷에서 채널 상태를 저장하는 과정에서 통신 비용을 크게 낮출 수 있다 [GGS10].

전역 스냅샷은 시스템을 구성하는 노드들의 내부 상태와 노드들 간의 채널 상태들의 집합으로써 정의된다. 일반적으로 전역 스냅샷을 만들기 위해 각 노드들은 다음과 같은 작업을 수행한다.

- 처음 모든 노드의 색깔은 흰색 이다.
- 흰색 노드는 흰색 메시지를 보내고, 붉은색 노드는 붉은색 메시지를 보낸다.
- 처음 전역 스냅샷 저장을 시작하는 노드는 다음과 같은 작업을 한다. 먼저

지역 스냅샷을 (local snapshot) 저장하고 색상을 붉은색으로 바꾼다. 그리고 자신과 연결된 모든 노드에게 "초기화"라는 메시지를 보낸다.

- 한 노드가 "초기화" 메시지를 받거나 붉은색 메시지를 받은 경우 자신의 색깔이 흰색이라면, 지역 스냅샷을 저장하고 색깔을 붉은색으로 바꾼 후 자신과 연결된 다른 노드들에게 "초기화" 메시지를 보낸다.

위와 같은 알고리즘을 시행하면 처음에 모두 흰색이었던 노드들이 "초기화" 메시지가 전파됨에 따라 차츰 붉은색으로 바뀌어 간다. 각 노드는 색상을 바꾸기 전에 지역 스냅샷을 저장하므로 어떤 노드가 붉은색으로 바뀌었다는 것은 해당 노드가 자신의 내부 상태들을 지역 스냅샷으로써 저장했음을 의미한다.

이렇게 노드들이 지역 스냅샷을 저장하는 과정에서 색상이 바뀌므로, 지역 스냅샷을 저장하기 전에 전송한 메시지는 흰색이고, 저장한 후에 보낸 메시지들은 붉은색이 된다. 모든 노드들이 자신들의 지역 상태를 저장한 다음에는 채널 상태들을 저장해야 한다.

예를 들어 n_i가 n_j에게 흰색 메시지 10개를 보낸 다음 지역 스냅샷을 저장했다고 하자. n_j가 10개의 메시지 중 5개를 수신한 다음 자신의 지역 스냅샷을 저장했다고 하자. 이미 수신한 5개 메시지의 정보는 n_j의 내부 상태에 반영되었으므로 n_j의 지역 스냅샷을 저장하는 것으로 이들 5개 메시지의 정보는 전역 스냅샷에 이미 반영 되었다. 그러나 남아있는 5개의 메시지는 이미 n_j가 지역 스냅샷을 저장했으므로 n_j의 지역 스냅샷을 통해서 저장될 수 없다. 이런 경우 남아 있는 5개의 흰색 메시지들이 채널 상태로써 저장되어야 한다.

분산 트리거 계수를 활용한 전역 스냅샷 알고리즘에서는 모든 노드의 지역 스냅샷을 저장한 후 채널 상태로써 저장되어야 하는 흰색 메시지의 수가 w라고 한다. 분산 트리거 계수 알고리즘을 사용하면 이 w개의 메시지가 모두 도착한 시점을 알

수 있고, 이 시점에 전역 스냅샷 저장을 완료할 수 있다.

이처럼 효율적인 분산 트리거 계수 알고리즘은 효율적인 대규모 분산 시스템을 위한 전역 스냅샷 알고리즘의 주요 부분으로써 응용이 가능하다.

1.4 연구 목적 및 범위

본 논문에서는 분산 트리거 계수 문제를 위한 두 가지 알고리즘 *TreeFill*과 *TreeFill-p*를 제시한다. 이들 알고리즘은 기존 연구와 같이 분산 시스템을 구성하는 노드들이 언제나 서로 통신할 수 있는 완전 그래프 형태의 토폴로지를 (topology) 이루고 있다고 가정한다 [GGS10, CCGS11, CCS11].

클라우드 컴퓨팅을 위한 하드웨어 플랫폼으로 사용되고 있는 데이터 센터의 경우 완전 그래프 형태의 노드 토폴로지를 만족한다. 노드들은 고속의 통신망으로 상호 연결되어 있으며 서로 통신 가능하다. 또한 데이터 센터의 각 노드는 비교적 신뢰성이 있으며 노드에서 실패가 (failure) 발생한 경우에도 재부팅 등을 통해 시스템에 곧 다시 참여한다 [ZCB10]. 따라서 데이터 센터의 경우 노드들이 트리와 같은 고정적인 토폴로지를 구성하며 루트 노드 역시 미리 선출한다고 가정할 수 있다.

하지만 완전 그래프를 가정할 수 없는 센서 네트워크나 애드혹 네트워크 (ad hoc network) 등은 제안 알고리즘이 가정하는 완전 그래프 형태의 토폴로지를 만족하지 않는다. 따라서 이들 시스템에 제안 알고리즘을 사용하려면 보다 일반적인 네트워크 토폴로지에서 효율적으로 작동하는 분산 트리거 계수 알고리즘을 설계해야 한다. 이 주제는 후속 연구에서 다루기로 한다.

본 논문에서는 *TreeFill* 및 *TreeFill-p*의 성능에 대한 증명을 제시하며 에이전트 기반 시뮬레이션 (agent based simulation) 도구인 NetLogo를 사용한 시뮬레이션

결과도 함께 제시한다.

1.5 연구 성과 요약

Garg는 노드의 수가 n이고 트리거의 수가 w라고 할 때, 분산 트리거 계수 문제를 해결하는 모든 정확한 알고리즘들의 (exact algorithms) 메시지 복잡도의 하한은 $\Omega(n \log(w/n))$임을 증명하였다 [GGS10]. 시스템에 참여하는 모든 노드가 균등하게 메시지 오버헤드를 처리한다고 할 때, MaxRcv의 하한은 $\Omega(\log(w/n))$ 이다.

본 논문에서 제시하는 *TreeFill* 알고리즘은 높은 확률로 Garg가 증명한 메시지 하한을 만족시키는 정확한 알고리즘이다. 즉, *TreeFill* 알고리즘은 높은 확률로 $O(n \log(w/n))$의 메시지 복잡도를 보인다. 또한 *TreeFill*의 MaxRcv는 $O(\log(w/n))$ 이다.

본 논문에서 제시하는 *TreeFill-p*는 확률적 알고리즘 (probabilistic algorithm) 이다. *TreeFill-p*는 정확한 알고리즘인 *TreeFill*과 달리 w개의 트리거가 n개의 노드에 검출된 경우에도 낮은 확률로 사용자에게 이를 알리지 않을 수 있다.

그러나 *TreeFill-p*는 어떤 상수 $m > 0$에 대해 $w = O(n^m)$인 경우 높은 확률로 $O(n)$의 메시지 복잡도를 보인다. *TreeFill-p*의 MaxRcv는 $O(1)$ 이다.

표 1.1는 기존의 분산 트리거 계수 알고리즘들과 본 논문에서 제안하는 *TreeFill* 및 *TreeFill-p* 알고리즘의 성능을 비교하여 보여준다.

Algorithm	Message Complexity	MaxRcv
Centralized [GGS10]	$O(n \log(w/n))$	$O(n \log(w/n))$
Tree-based [GGS10]	$O(n \log n \log(w/n))$	$O(n \log n \log(w/n))$
LayeredRand [CCGS11]	$O(n \log n \log w)$	$O(\log n \log w)$
CompTreeRand [EK10]	$O(n \log w (\log \log n)^2)$	–
CompTreeDet [EK10]	$O(n(\log w \log n)^2)$	$O((\log w \log n)^2)$
CoinRand [CCS11]	$O(n(\log w + \log n))$	$O(\log w + \log n)$
RingRand [CCS11]	$O(n \log n \log w)$	$O(\log n \log w)$
TreeFill	$O(n \log(w/n))$	$O(\log(w/n))$
TreeFill-p	$O(n)$	$O(1)$

(Emek의 알고리즘은 임의의 네트워크에 대한 성능이다 [EK10]. *TreeFill-p*는 정확한 알고리즘이 (exact algorithm) 아닌 확률적 알고리즘 (probabilistic algorithm)이며 $m > 0$인 상수에 대해 $w = O(n^m)$이라는 추가 가정이 필요하다.)

표 1.1: 분산 트리거 계수 알고리즘들의 성능 비교.

1.6 논문의 구성

본 논문은 다음과 같이 구성된다. 2장에서는 기존 분산 트리거 계수 알고리즘들과 분산 트리거 계수 알고리즘을 직접적으로 응용할 수 있는 전역 스냅샷 알고리즘 기법들을 논의한다. 본 논문에서 제안하는 *TreeFill* 알고리즘을 3장에서, *TreeFill-p* 알고리즘을 4장에서 설명한다. 각 알고리즘의 성능 역시 해당 장에서 분석한다. 5장에서는 NetLogo를 사용하여 *CoinRand* 알고리즘과 본 논문에서 제시하는 *TreeFill* 및 *TreeFill-p* 알고리즘의 성능을 비교 한다. 마지막으로 6장에서 결론을 맺는다.

제 2 장 관련 연구

기존에 제시된 분산 트리거 계수 알고리즘들은 2.1절에서 논의한다. 분산 트리거 계수 알고리즘은 대규모 분산 시스템을 위한 전역 스냅샷 알고리즘에 직접적으로 응용이 가능하다. 대규모 분산 시스템을 위해 제시된 전역 스냅샷 알고리즘들은 2.2절에서 논의한다.

2.1 분산 트리거 계수 알고리즘

2.1.1 Garg의 분산 트리거 계수 알고리즘들과 분산 트리거 계수 문제의 메시지 복잡도 하한

Garg는 분산 트리거 계수 문제를 위한 세 가지 알고리즘을 제시하고 이 알고리즘들을 함께 사용할 수 있는 방법을 제시 하였다. Garg에 의하여 제안된 알고리듬은 그리드 기반, 트리 기반, 중앙 집중식 알고리즘이다 [GGS10]. Garg는 또한 분산 트리거 계수 알고리즘을 위한 하한을 제시 하였다 [GGS10]. 분산 트리거 계수 문제를 위한 결정론적 알고리즘의 (deterministic algorithm) 메시지 복잡도의 하한은 $\Omega(n \log(w/n))$ 이다. Garg의 알고리즘에서 노드들 사이에 주고받는 메시지의 크기는 $O(1)$ 이다. 이는 아래에서 논의하는 다른 분산 트리거 계수 알고리즘들도 마찬가지이다.

Garg가 제안한 중앙 집중 방식의 분산 트리거 계수 알고리즘의 메시지 복잡도는 $O(n(\log w/n))$ 이다. 이는 최적의 메시지 복잡도 이다. 그러나 중앙 집중 방식

알고리즘은 하나의 마스터 노드가 모든 메시지를 모으는 방식이기 때문에 MaxRcv 역시 $O(n(\log w/n))$가 된다. Garg의 트리 기반 알고리즘은 라운드 기반으로 작동한다. 각 라운드에서 트리거들을 검출하기 위해 이진 트리를 사용한다. 각 라운드가 시작할 때 이진 트리의 각 노드는 트리거에 의해 소비될 수 있는 토큰을 (tokens) 받게 된다. 트리 기반 알고리즘에서 이진 트리의 역할은 트리거들이 소비할 수 있는 토큰들을 찾는 것이다. 각 라운드에서 트리거들이 모든 토큰을 소비하게 되면 트리 기반 알고리즘의 한 라운드가 끝난다. 트리기반 알고리즘의 메시지 복잡도와 MaxRcv는 모두 $O(n \log n \log(w/n))$ 이다.

2.1.2 *LayeredRand* 알고리즘

Chakaravarthy는 *LayeredRand* 알고리즘을 제시하였다 [CCGS11]. 이 알고리즘의 메시지 복잡도는 $O(n \log n \log w)$ 이며, MaxRcv는 $O(\log n \log w)$ 이다 [CCGS11]. *LayeredRand* 알고리즘에서 시스템의 노드들은 이진 트리와 유사한 계층 구조를 형성한다. 전체 노드 수 n은 어떤 정수 L에 대하여 $n = 2^L - 1$ 이다. 각 노드들은 계층0에서 계층$L-1$를 형성하며 계층 i에 $(0 \leq i < L)$ 있는 노드의 수는 2^i이다.

*LayeredRand*은 여러 라운드에 걸쳐서 이루어진다. w개의 트리거를 검출하기 위해 *LayeredRand* 알고리즘의 각 라운드는 아직 검출되지 않은 트리거의 수를 알고 있어야 한다. 이는 이전 라운드까지 검출된 트리거의 수와 w의 차이를 통해 알 수 있다. 각 라운드에서 아직 검출되지 않은 트리거의 수를 $\hat{w}$라 하자.

*LayeredRand*의 각 노드 x는 자신이 검출한 트리거의 수 $C(x)$를 가지고 있다. *LayeredRand*의 각 라운드에서 계층l에 있는 노드는 다음의 $\tau(l)$개의 트리거를 검출한 경우, 계층 $l-1$에 있는 하나의 노드 y를 무작위로 선택하여 메시지를 보낸다.

이 메시지를 코인이라 부르기로 한다.

$$\tau(l) = \left\lceil \frac{\hat{w}}{4 \cdot 2^l \cdot \log(n+1)} \right\rceil.$$

노드 y가 코인을 하나 받으면 $C(y)$를 $\tau(l)$만큼 증가 시킨다. 만약 $C(y) \geq \tau(l-1)$을 만족하게 되면 무작위로 계층 $l-2$에 있는 노드 z를 선택하고 코인을 보낸다. 코인을 보낸 후 $C(y)$를 $\tau(l-1)$만큼 감소 시킨다. 노드 z 역시 마찬가지로 동작한다.

루트 노드 r역시 코인을 보내지 않는 다는 점을 제외하고 다른 노드와 마찬가지로 동작한다. 루트 노드의 중요한 과제는 각 라운드를 언제 끝낼지 판단하는 것이다. 루트 노드 r의 카운트 $C(x) \geq \lceil \hat{w}/2 \rceil$을 만족하면 루트 노드는 해당 라운드가 끝났다고 판단한다. 이제 루트는 현재 라운드에서 모든 노드들이 수신한 트리거들의 정확한 수를 파악해야 한다. 이를 위해 미리 노드들 사이에 이진 트리가 구축되어 있다고 가정한다. 이 트리를 이용하여 전체 노드들이 수신한 트리거의 수를 집계한다. 이 때 필요한 메시지의 수는 $O(n)$이다.

*LayeredRand*의 각 라운드는 $O(n \log n)$개의 코인을 사용한다. *LayeredRand*에게 필요한 라운드의 수는 $O(\log w)$이다. 그래서 *LayeredRand* 알고리즘의 메시지 복잡도는 $O(n \log \log w)$가 된다. 이 때 MaxRcv는 $O(\log n \log w)$가 된다.

2.1.3 *CoinRand*, *RindRand* 알고리즘

Chakaravarthy는 *CoinRand*과 *RindRand* 알고리즘을 또한 제안하였다 [CCS11]. *CoinRand* 알고리즘은 *LayeredRand* 알고리즘과 유사하지만 보다 향상된 메시지 복잡도와 MaxRcv를 가지고 있다. *RindRand* 알고리즘은 다른 분산 트리거 계수 알고리즘이 고려하지 않은 MaxSnd를 고려하였다. MaxSnd는 MaxRcv에 대응되

는 개념으로써, 시스템에서 가장 많은 메시지를 송신한 노드가 보낸 메시지의 총 수이다.

CoinRand 알고리즘의 메시지 복잡도는 $O(n(\log w + \log n))$ 이며 MaxRcv는 $O(\log w + \log n)$ 이다. *RindRand* 알고리즘의 메시지 복잡도와 MaxRcv는 각각 *LayeredRand*과 같은 $O(n \log n \log w)$, $O(\log n \log w)$ 이지만, 다른 분산 트리거 계수 알고리즘과는 달리 추가적으로 $O(\log n \log w)$의 MaxSnd를 보장한다.

이 절에서는 *CoinRand* 알고리즘에 대하여 자세하게 살펴본다. *CoinRand*은 *LayeredRand* 알고리즘과 유사한 계층 구조를 사용한다. 어떤 정수 L에 대하여 $n = 2^L$이라 가정하자. *CoinRand* 알고리즘은 $L + 1$ 개의 계층으로 이루어져 있다. 계층 i에는 $(0 \leq i \leq L)$ 2^i개의 노드가 있다.의 가장 아래 계층인 계층 L에는 2^L개의 노드가 있다. $n = 2^L$이므로 전체 노드가 계층 L에 존재한다. 계층 0 $L - 1$에 존재하는 노드의 수는 $n - 1$이다. 따라서 전체 노드 중 $n - 1$개의 노드는 계층 L과 다른 또 다른 계층에 속해 있다. 즉, 하나의 노드가 두 계층에 속해서 두 가지 역할을 하는 것이다.

CoinRand 알고리즘 역시 라운드 기반으로 동작한다. *LayredRand* 알고리즘과 마찬가지로 $\hat{w}$를 각 라운드가 시작할 때 아직 검출되지 않은 트리거의 수라 하자. 각 노드는 임계값 $\tau = \lceil \hat{w}/4n \rceil$을 계산한다. 각 노드 x는 자신이 검출한 트리거의 수를 나타내는 $C(x)$를 또한 가지고 있다. 노드 x가 트리거를 하나 받으면 $C(x)$를 1 증가 시킨다. 노드 x가 계층 l에 있다고 하자. 만약 $C(x)$가 τ에 다다르면 계층 $l - 1$에서 무작위로 노드 y를 선택하여 코인을 보내고 $C(x)$를 0으로 재설정 한다. 만약 이것이 노드 y가 첫번째로 받은 코인이라면 특별한 추가 동작을 하지 않는다. 만약 이것이 두번째로 받은 코인이라면 노드 y의 바로 위 계층에서 무작위로 노드 z를 선택하여 코인을 보낸다. 노드 z도 유사하게 동작한다.

*CoinRand*에서 루트 노드는 다른 노드들과 다르게 동작한다. 루트 노드가 코인을 받으면 현재 라운드에서 검출한 정확한 트리거의 수를 미리 정의된 이진 트리를 통하여 집계한다. 현재 라운드에서 검출한 트리거의 수 w'는 $C_{sum} = \sum C(x)$ 이고 N_{coins}가 모든 노드가 가지고 있는 코인들의 총 합이라 할 때 $w' = C_{sum} + \tau \times N_{coins}$가 된다. 그러면 새로운 $\hat{w}$는 $\hat{w} = \hat{w} - w'$로 계산할 수 있다. 이렇게 새로운 $\hat{w}$를 계산하여 새로운 라운드를 시작하게 된다.

이러한 방식으로 트리거를 검출하는데 필요한 라운드의 수는 $O(\log w)$이다. 각 라운드에서 노드들은 하나의 코인만을 바로 위 계층으로 보내고, 각 노드는 계층 L 및 또 다른 계층에 속할 수 있으므로 최대 2개의 코인을 보낸다. 따라서 각 라운드에서 사용하는 메시지의 수는 $O(n)$이 되며 메시지 복잡도는 $O(n \log w)$가 된다.

계속 트리거를 검출하다 보면 $\hat{w} < n$인 라운드가 오게 된다. 이 때는 남아 있는 트리거의 수가 노드의 수 보다 작기 때문에 전체 노드가 트리거 검출에 참여할 필요가 없다. 그러면 계층 구조의 높이를 줄인다. 줄어든 계층 구조에 참여하는 노드 수가 n'라 하자. 그러면 $\hat{w} > n'$가 될 때 까지 계층 구조의 높이를 줄이고 위에서 설명한 알고리즘을 작동한다. 이 때 필요한 라운드의 수는 $O(\log n)$이다. 따라서 *CoinRand*의 메시지 복잡도는 $\hat{w} \geq n$인 경우와 $\hat{w} < n$인 경우를 합하여 $O(n(\log w + \log n))$이 된다. *CoinRand* 알고리즘에서 각 노드는 $O(1)$개의 메시지를 수신하기 때문에 *CoinRand*의 MaxRcv는 알고리즘에 필요한 라운드의 수인 $O(\log n + \log w)$가 된다.

2.1.4 트리 네트워크에서의 분산 트리거 계수 알고리즘

Emek과 Korman은 보다 일반화된 네트워크 구조를 위한 분산 트리거 계수 알고리즘을 제시하였다 [EK10]. 앞에서 설명한 분산 트리거 계수 알고리즘들은 시스템에 참여하는 모든 노드들이 메시지를 주고 받을 수 있다고 가정하고 알고리즘을 설

계하였다. 그러나 Emek과 Korman은 트리 네트워크를 가정하고 오직 인접 노드들 사이에서만 메시지를 주고 받을 수 있다고 가정 하였다. 이들이 제안한 두 알고리즘을 Chakaravarthy가 명명한 데로 *CompTreeRand*, *CompTreeDet*라 하자 [CCS11]. *CompTreeRand*의 메시지 복잡도는 $O(n \log w (\log \log n)^2)$이고 MaxRcv는 상한을 가지지 않는다. *CompTreeDet*의 메시지 복잡도는 $O(n(\log w \log n)^2)$이며 MaxRcv는 $O((\log w \log n)^2)$ 이다.

2.2 대규모 분산 시스템을 위한 전역 스냅샷 알고리즘

분산 트리거 계수 알고리즘은 쉽게 대규모 분산 시스템을 위한 전역 스냅샷 알고리즘에 사용 가능하다. Garg는 분산 메시지 계수 문제를 (distributed message counting problem) 정의하였다 [GGS10]. 이 정의는 용어만 다를 뿐 분산 트리거 계수 문제의 정의와 같다.

최근의 대규모 분산 시스템은 수천개의 컴퓨팅 노드들로 구성되어 있다 [ss12]. 전역 스냅샷은 이러한 대규모 분산 시스템에서 중요한 기능이다 [SBF+04, CHL+06, ORS06, KXRE07, GGS10].

분산 시스템의 전역 스냅샷은 시스템을 구성하는 노드들의 상태와 (일반적으로 프로세서들의 상태, processor states) 채널들의 상태의 (channel states) 집합이다 [CL85]. 만약 노드들이 스패닝 트리를 (spanning tree) 구성하고 있는 경우 노드들의 상태를 저장하기 위해서는 노드의 수가 n이라 할 때 (n)개의 메시지가 필요하다. 그러나 일반적으로 채널들의 상태를 저장하기 위해서는 $O(n^2)$의 메시지들이 필요하다 [CL85, LY87, Mat93]. 만약 분산 시스템에 참여하는 노드의 수가 수천개 이상으로 증가할 경우 이는 감당하기 어려운 오버헤드가 된다.

분산 트리거 계수 알고리즘은 전역 스냅샷 알고리즘의 채널 상태 저장 단계에

이용 가능하다 [GGS10]. 분산 트리거 계수 알고리즘을 사용함으로써 분산 스냅샷 알고리즘의 채널 상태 저장 비용을 크게 줄일 수 있다.

Kshemkalyani는 하이퍼큐브 기반의 분산 스냅샷 알고리즘을 제안 하였다. 이 알고리즘은 분산 트리거 계수를 이용한 전역 스냅샷 알고리즘들이 크기 $O(1)$인 메시지를 사용하는데 반하여 크기 $O(n)$짜리 메시지를 사용한다. 하지만 알고리즘의 응답 속도가 $O(n \log n)$으로 우수하다. 반면 분산 트리거 계수 알고리즘은 최악의 경우 응답 시간이 알고리즘의 메시지 복잡도와 같아 진다. 예를 들어 *CoinRand* 알고리즘의 경우 응답 속도는 자신의 메시지 복잡도인 $O(n(\log w + \log n))$이 된다. 하이퍼 큐브 기반 알고리즘이 사용하는 메시지의 총 수는 $O(n \log n)$ 이다.

최근에 Tsai는 하이퍼뷰브 기반 분산 스냅샷 알고리즘을 일반화하여 일반적인 그리드 형태의 연결 네트워크에서 설계하는 전역 스냅샷 알고리즘을 위한 메시지 복잡도의 하한을 증명하였다 [Tsa13].

제 3 장 분산 트리거 계수를 위한 최적 알고리즘

본 장에서는 분산 트리거 계수 문제를 높은 확률로 최적의 메시지 오버해드로 해결하는 *TreeFill* 알고리즘을 제안한다. Garg는 전체 노드의 수가 n이고 검출해야 하는 트리거의 수가 w일 때 노드 간에 주고 받는 콘트롤 메시지의 총 합은 $\Omega(n \log(w/n))$임을 증명 하였다 [GGS10]. *TreeFill*은 높은 확률로 $O(n \log(w/n))$개의 콘트롤 메시지를 사용하여 w개의 트리거를 검출할 수 있다. 3.1절에서는 *TreeFill* 알고리즘이 가정하고 있는 시스템 모델을 설명한다. 3.2절에서는 *TreeFill* 알고리즘을 기술한다. 마지막로 3.3절에서 *TreeFill* 알고리즘의 성능을 메시지 복잡도와 MaxRcv의 측면에서 분석 한다.

3.1 시스템 모델

전체 시스템은 n개의 노드, $p_1, p_2, \cdots, p_n$ 으로 이루어져 있다. 각 노드는 서로 메시지를 주고 받을 수 있다. 따라서 노드들의 연결은 완전 그래프를 형성하고 있다. 이 n개의 노드에게 w개의 트리거가 도착한다. 트리거들의 분포에 대한 어떠한 통계 정보도 미리 주어지지 않는다. 전체 시스템은 각 노드에 도착한 트리거들의 총 합이 w가 되는 경우 사용자에게 이를 보고해야 한다.

트리거는 시스템 내부 또는 외부의 특정 이벤트를 기준으로 응용에 맞게 정의할 수 있다. 예를 들어 데이터 센터에 현재 접속 중인 사용자의 수를 분산 트리거 계수 알고리즘으로 추적하는 경우, 시스템에 새로운 사용자가 접속하는 경우 이것이 트리거가 된다. 사용자가 시스템에서 로그아웃 하는 경우를 또 다른 트리거로 정의하고 시스템 로그인 트리거의 수에서 시스템 로그아웃 트리거의 수를 빼면 현재 접속중인 사용자 수를 계산할 수 있다. 또 다른 예로써 데이터 센터에서 각 노드의 cpu 및 메모리 사용량이 미리 정한 한계값을 넘어서는 경우 트리거를 발생 시킬 수 있다. 이런 경우 트리거의 수가 많아 지면 시스템 자원이 많이 사용되었다는 뜻이므로 데이터 센터 전체의 리소스 사용량을 유추하기 위해 분산 트리거 계수를 이용할 수 있다.

이렇게 다양한 이벤트에 대응하여 트리거를 발생시킬 수 있으므로, 트리거의 분포에 대해 어떠한 통계적 가정을 하기 어렵다. 그래서 분산 트리거 계수 문제에서 n개 노드에 도착하는 w개의 트리거에 대한 어떠한 통계 정보도 미리 주어지지 않는다고 가정한다. 또한 트리거의 수 w는 $w \gg n$이라고 가정한다. 만약 어떤 상수 c에 대해 $w = cn$이라면 발생하는 모든 트리거를 특정한 마스터 노드에 전달하는 것만으로도 $O(n)$의 메시지 오버헤드를 가지는 알고리즘을 만들 수 있으므로 문제가 매우 간단해 진다.

3.2 *TreeFill* 알고리즘

TreeFill 알고리즘은[1] 라운드 기반으로 작동하여 n개의 노드로 w개의 트리거를 분산 검출하게 된다. *TreeFill*의 각 라운드에서는 이전 라운드들을 통해 검출 되지 않은 트리거들의 절반을 검출한다. 예를 들어 첫번째 라운드에서 검출되지 않은 트리거의 수는 w이므로 첫번째 라운드에서는 $w/2$개의 트리거를 검출한다. 두 번째 라운드에서는 남은 트리거의 반인 $w/4$개의 트리거를 검출한다.

TreeFill 알고리즘에 참여하는 노드들은 두 가지 역할을 수행할 수 있다. 하나는 각 노드에서 일정 수의 트리거를 검출한 경우 콘트롤 메시지를 새로 만든다. 이렇게 만들어진 콘트롤 메시지들을 취합하여 전체 시스템에 도착한 트리거의 수를 알 수 있다. 이후로 이렇게 일정한 트리거를 검출한 경우 만들어 내는 콘트롤 메시지를 코인이라 (coin) 부르기로 한다. 노드들의 또 다른 역할은 *DetectTree*라 부르는 노드들의 이진 트리를 구성하여 노드들에게서 발생한 코인의 수가 n이 되는 순간을 감지해 내도록 하는 것이다.

첫번째 라운드가 시작할 때 앞으로 검출해야 할 트리거의 수는 w이다. i번째 라운드가 시작할 때에 앞으로 검출해야 할 트리거의 수를 w_i라 하자. *TreeFill* 알고리즘의 i번째 라운드에서 각 노드는 $\tau_i = w_i/2n$개의 트리거를 검출하는 경우 하나의 코인을 만든다. 그러면 트리거의 도착 분포와 관계 없이 항상 최소한 n개의 코인이 발생하게 된다. 다음 정리 1는 이를 보여준다.

정리 1 *TreeFill 알고리즘의 i번째 라운드에서 $w_i > 2n$인 경우 각 노드는 $\tau_i = w_i/2n$ 개의 트리거를 받을 때 마다 하나의 코인을 발생 시킨다. 그러면 해당 라운드에서 트리거의 분포와 관계 없이 항상 최소한 n개의 코인이 발생한다.*

[1] *TreeFill* 알고리즘은 [KLPC13]에 발표 되었음.

증명 i번째 라운드에서 $w_i > 2n$인 경우, 각 노드는 코인을 발생시키지 않고 $(w_i/2n)-1$개의 트리거를 검출할 수 있다. 따라서 코인을 하나도 발생시키지 않고 전체 노드가 검출할 수 있는 트리거 수의 최대값은 $n \cdot ((w_i/2n) - 1) = w_i/2 - n$이다. 이 경우 남아 있는 트리거의 수는 $w_i - (w_i/2 - n) = w_i/2 + n$ 이다.

이 경우 코인을 발생시킨 노드에 $(w_i/2n) - 1$개의 트리거가 도착하면 코인을 발생시키지 않을 수 있다. 그 다음에 도착하는 트리거는 모든 노드가 이미 $(w_i/2n) - 1$개의 트리거를 받은 상황이기 때문에 어떤 노드에 트리거가 도착하더라도 하나의 코인이 발생한다. 따라서 최소한 $w_i/2 + n$개의 트리거들은 $w_i/2n$개의 트리거가 도착할 때 마다 하나의 코인을 발생 시킨다. 이 경우 발생하는 코인 수는 다음과 같다

$$\lfloor (w_i/2 + n)/(w_i/2n) \rfloor = \lfloor n + 2n^2/w_i \rfloor \geq n.$$

위와 같은 경우보다 더 적은 수의 코인을 발생 시킬 수는 없다. w개의 트리거가 도착 했으나 $m < n$인 m개의 코인이 발생했다고 가정하자. 그러면 $m \cdot w_i/2n$개의 트리거들이 m개의 코인을 만드는데 필요하다. $m/n < 1$이므로 이는 $w_i/2$보다 작은 값이다. 따라서 $w_i/2$ 보다 많은 트리거들은 코인을 만들지 않고 전체 노드에 의해 검출될 수 있어야 한다. 그러나 코인을 만들지 않고 모든 노드들이 검출할 수 있는 트리거 수의 최대 값은 $w_i/2-n$이므로 이는 모순이다. 따라서 트리거의 분포와 관계없이 $w_i > 2n$을 만족하는 라운드에서는 최소한 n개의 코인이 발생한다.

*TreeFill*의 각 라운드들은 $w_i > 2n$인 경우와 $w_i \leq 2n$인 경우 작동 방식이 다르다. $w_i \leq 2n$인 경우는 뒤에서 자세히 다루기로 한다.

*TreeFill*의 i번째 라운드에서 $w_i > 2n$인 경우, 최소한 n개의 코인이 발생하므로 이 n개의 코인을 검출하면 $n\tau_i = w_i/2$개의 트리거가 전체 시스템에 도착했음을 알

수 있다. 따라서 $w_i > 2n$를 만족하는 i번째 라운드에서 n개 코인을 검출하면 $w_i/2$개의 트리거들이 시스템에 도착했음을 알 수 있다.

첫번째 라운드에서는 $w_1 = w$이므로 $w/2$개의 트리거가 검출된다. 두번째 라운드에서는 n개의 코인을 검출하면 $w/4$개의 트리거가 검출된다. 따라서 $w_i > 2n$인 경우 i번째 라운드에서 n개의 코인을 통해 검출 가능한 트리거의 수는 $w/2^i$ 이다.

TreeFill 알고리즘의 각 라운드에서 n개의 코인을 검출하기 위해 다음 그림 3.1과 같은 노드들의 이진 트리를 사용한다. 이 이진 트리를 *DetectTree*라 부르기로 한다.

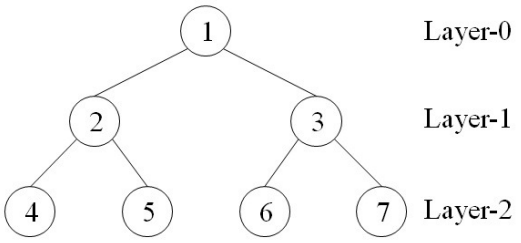

그림 3.1: $n = 2^{2+1} = 8$인 경우 *DetectTree*의 예.

설명의 편의를 위해 전체 노드의 수 n은 어떤 정수 h에 대해 $n = 2^{h+1}$이라 가정한다. (n이 2^{h+1}이 아닌 경우에 대해서도 쉽게 *TreeFill* 알고리즘을 확장할 수 있다.)

*DetectTree*의 높이는 h이다. 따라서 *DetectTree*에 참여하는 노드의 수는 $2^{h+1} - 1 = n - 1$이다. 전체 n개의 노드 중 $n - 1$개는 $w_i/2n$개의 트리거를 받았을 때 하나의 코인을 만드는 역할과, *DetectTree*의 한 노드로써 n개의 코인을 분산 검출하는 작업에도 참여 한다. 남아있는 한 개의 노드는 코인을 만들어 내는 역할만 수행한다.

알고리즘 3.1은 각 노드 p_i에서 τ_i개의 트리거를 받았을 때 코인을 만들고 *DetectTree*에 전송하는 알고리즘을 나타낸다.

정리 1에서 보인 바와 같이 $w_i > 2n$을 만족하는 i번째 라운드에서는 항상 최소

알고리즘 3.1 노드 p_i가 $\tau_i = w_i/2n$ 개의 트리거를 받았을 때 코인을 *DetectTree*에 전송하는 알고리즘

첫번째 라운드가 시작할 때, 각 노드 p_i는 다음을 수행:

$p_i.trgs \leftarrow 0$.

i번째 라운드에서 각 노드 p_i는 다음을 수행:

p_i가 하나의 트리거를 받았을 때:

$p_i.trgs \leftarrow p_i.trgs + 1$.

만약 $p_i.trgs \geq w_i/2n$인 경우:

*DetectTree*의 리프 노드 중 하나인 f를 무작위로 선택.

코인 하나를 f에게 전송.

$p_i.trgs \leftarrow p_i.trgs - w_i/2n$.

n개의 코인이 발생한다. *DetectTree*는 이 n개의 코인을 분산 검출한다.

*DetectTree*의 코인 분산 검출 알고리즘을 설명하기 위해 다음과 같이 *DetectTree*에서의 i번째 계층을 정의한다.

정의 1 *DetectTree에서 루트 노드가 높이 0에 있다고 할 때 높이 i에 있는 모든 노드를 DetectTree의 i번째 계층이라고 정의한다.*

*DetectTree*의 i번째 계층에 있는 노드들의 수는 2^i가 되며 i는 $0 \leq i \leq h$을 만족한다.

*DetectTree*의 높이는 h이므로 리프 노드의 수는 $2^h = 2^{h+1}/2 = n/2$개 이다. 각 리프 노드는 2개의 코인을 받으면 *FULL* 메시지를 부모 노드에게 전송 한다. *DetectTree*의 내부 노드는 2개의 *FULL* 메시지를 받으면 자신의 부모 노드에게 *FULL* 메시지를 전송한다. 따라서 모든 리프 노드가 코인을 2개씩 가지도록 하면 리프 노드 전체가 받은 코인은 n개가 되고 모든 내부 노드가 자신들의 부모에게 *FULL*

메시지를 전송하게 되어 루트 노드가 결국 *FULL* 메시지를 받게 된다. 루트 노드는 자신이 *FULL* 메시지를 받으면 모든 리프 노드가 가지고 있는 코인들의 합이 n임을 알 수 있다.

전체 시스템을 구성하는 n개의 노드들은 코인을 전송할 경우 $n/2$개의 리프 노드 중 하나를 무작위로 선택하여 보내게 된다. 그러면 *DetectTree*의 리프 노드들이 수신하는 코인 수의 평균은 2이다. *DetectTree*는 평균보다 많은 수의 코인을 받은 노드에서 평균보다 적은 수의 코인을 받은 노드에게 효율적으로 코인을 전달해 줌으로써, n개의 코인이 $n/2$개의 리프 노드들에게 보내질 때 모든 노드들이 2개씩의 코인을 받도록 만들어 준다. 이를 통해 *DetectTree*는 n개의 코인을 분산 검출할 수 있다.

n개의 코인이 *DetectTree*에 도착할 때, 리프 노드는 첫번째 코인을 받았을 때는 별다른 추가 동작을 하지 않는다. 두번째 코인을 받으면 부모 노드에게 *FULL* 메시지를 보낸다. 세번째 이후로 받는 코인들은 모두 자신이 가지지 않고, 높이 $h-1$번째 계층의 노드 중 하나에게 무작위로 전달한다. *DetectTree*의 내부 노드들은 이렇게 리프 노드들에게서 전달 받은 코인들을 2개 미만의 코인을 가지고 있는 리프 노드에게 전달해 준다.

*DetectTree*의 내부 노드 r은 $r.full[1..2]$ 배열을 가지고 있다. 각 라운드가 시작할 때 모든 내부 노드의 $r.full[1..2]$는 *false*로 초기화 된다. 내부 노드의 좌측 자녀 노드가 *FULL* 메시지를 보내면 $r.full[1]$을, 우측 자녀 노드가 *FULL* 메시지를 보내면 $r.full[2]$가 *true*로 바뀐다. 만약 $r.full[1..2]$가 모두 *true*로 바뀌면 내부 노드의 부모 노드에게 *FULL* 메시지를 보낸다.

내부 노드 r이 코인을 전달 받은 경우 $r.full[1..2]$ 배열을 이용하여 2개 미만의 코인을 받은 리프 노드에게 코인을 전달해 줄 수 있다. 만약 내부 노드 r이 코인을

전달 받았는데, $r.full[1..2]$ 중 하나가 $false$라면 r의 서브 트리 중 하나에 2개 미만의 코인을 가진 노드가 최소한 하나가 있음을 의미한다. 따라서 코인을 해당 서브 트리에게 전달하고, 같은 방식으로 재귀적으로 코인을 전달하여 2개 미만의 코인을 가진 리프 노드에게 코인을 전해줄 수 있다. 만약 $r.full[1..2]$가 모두 $true$라면 이는 r을 루트 노드로 하는 서브 트리의 모든 리프 노드는 2개의 코인을 가지고 있음을 의미한다. 따라서 r의 높이가 i라면, $r.full[1..2]$가 모두 $true$이고 코인을 전달 받은 경우에 *DetectTree*의 $i-1$번째 계층에 있는 노드 중 하나에게 코인을 무작위로 전달한다. 이 과정 역시 재귀적으로 계속 진행될 수 있다. 이상과 같은 방법을 통해 *DetectTree*는 n개의 코인을 검출할 수 있다. 알고리즘 3.2는 *DetectTree*에서 n개 코인을 검출하는 알고리즘을 보여준다.

알고리즘 3.2를 사용하여 트리거들을 검출하면 $w_i \leq 2n$인 라운드에 도달하게 된다. 이 경우 $\tau_i = 1$이 되며, 이는 모든 노드들이 하나의 트리거를 받으면 하나의 코인을 만들어서 *DetectTree*의 리프 노드들 중 하나에게 무작위로 보냄을 뜻한다. *DetectTree*는 n개의 코인을 검출할 수 있으므로 남아 있는 트리거의 수가 $w_i \leq 2n$를 만족하면 최대 2라운드가 더 필요하게 된다.

i번째 라운드가 시작할 때 남이 있는 트리거의 수가 $n \leq w_i \leq 2n$을 만족하는 경우에는 τ_i를 1로 설정하는 것 외에 특별히 더 필요한 설정은 없다. 그러나 $w_i < n$이 되는 경우에는 *DetectTree*는 n개의 코인을 받아야 종료할 수 있는데 코인 수가 부족하므로 해당 라운드가 종료될 수 없다. 이런 경우를 막기 위해 i번째 라운드가 시작할 때 $w_i < n$이면 $n - w_i$개의 코인을 임의로 미리 *DetectTree*의 리프 노드들에게 배포한다. 그러면 남아 있는 w_i개의 코인이 도착하면 미리 배포했던 코인을 포함하여 *DetectTree*가 수신한 코인의 총 수가 $n - w_i + w_i = n$이므로 해당 라운드가 종료될 수 있다. $w_i \leq n$이 되면 남아 있는 모든 트리거를 검출할 수 있으므로 *TreeFill* 알고리즘이 종료되고 w개의 트리거를 분산 검출하는 작업을 끝내게 된다.

알고리즘 3.2 *DetectTree*에서 n개의 코인을 분산 검출하는 알고리즘

각 라운드가 시작할 때 내부 노드 r은 다음을 수행:
  배열 $r.full[]$ 전체를 $false$로 초기화.

*DetectTree*의 리프 노드 f는 다음을 수행:
  f가 코인을 받은 경우:
    $f.coins \leftarrow f.coins + 1$.
    $f.coins$가 처음 2가 된 경우:
      $FULL$ 메시지를 부모에게 전송.
    If $f.coins > 2$ then:
      코인을 바로 위 계층의 노드 중 하나에게 무작위로 전달.
      $f.coins \leftarrow f.coins - 1$.

*DetectTree*의 내부 노드 r은 다음을 수행:
  r이 $r.child[i]$에게 $FULL$을 받은 경우: //i=1이면 왼쪽, i=2이면 오른쪽
    $r.full[i] \leftarrow true$.
    만약 배열 $r.full[]$ 내부가 모두 $true$인 경우:
      $FULL$ 메시지를 부모에게 전송.
  r이 코인을 받은 경우:
    $r.full[i] = false$을 만족하는 i 찾기.
    만약 이를 만족하는 i가 없다면:
      코인을 바로 위 계층의 노드 중 하나에게 무작위로 전달.
    그렇지 않다면:
      코인을 $r.child[i]$로 전달.

*DetectTree*의 루트 노드는 다음을 수행:
  다른 내부 노드와 같이 코인을 전달하거나 $r.full[]$ 배열 업데이트.
  만약 배열 $r.full[]$ 전체가 모두 $true$ 라면:
    $w_i/2$개의 트리거를 검출 하였음.
    다음 라운드 시작.

3.3 *TreeFill* 알고리즘 성능 분석

이 절에서는 *TreeFill* 알고리즘의 메시지 복잡도 및 MaxRcv를 분석한다. 먼저 3.3.1 절에서 *TreeFill* 알고리즘이 종료될 때 까지 필요한 라운드 수가 $O(\log(w/n))$이고, *DetectTree*가 각 라운드에서 n개 코인을 분산 검출하기 위해 노드들 간에 주고받는 메시지의 총 수가 높은 확률로 $O(n)$이므로 전체 *TreeFill* 알고리즘의 메시지 복잡도가 높은 확률로 $O(n\log(w/n))$임을 증명한다. 다음으로 3.3.2절에서 각 라운드에서 코인의 분산 검출을 위해 노드들이 수신하는 메시지의 수가 높은 확률로 $O(1)$이고 *TreeFill*에 필요한 라운드 수는 $O(\log(w/n))$이므로 *TreeFill* 알고리즘에서 MaxRcv는 높은 확률로 $O(\log(w/n))$ 임을 증명한다.

3.3.1 *TreeFill* 알고리즘의 메시지 복잡도

먼저 *TreeFill*알고리즘이 필요로 하는 라운드의 수가 $O(\log(w/n))$임을 증명 한다. $\tau_i = w_i/2n > 1$인 경우 필요한 라운드 수를 생각해 보자. *TreeFill*의 각 라운드는 $w_i/2$의 트리거를 검출하므로 $w_i = w/2^{i-1}$ 이다. $tau_i \leq 1$이 되는 경우의 i는 다음과 같이 알 수 있다.

$$\tau_i = w_i/2n \leq 1$$
$$w/2^{i-1} \leq 2n$$
$$\log(w/n) \leq i$$

즉, $j = \lceil \log(w/n) \rceil$ 번째 라운드가 되면 $w_j/2n \leq 1$을 만족하며 $\tau_j = 1$이 된다. 남아 있는 트리거의 수가 $2n$보다 작거나 같고 $\tau_j = 1$이며 *DetectTree*는 n개의 코인을 분산 검출 할 수 있으므로 최대 2라운드가 더 있으면 w개의 트리거를 모두 분산

검출할 수 있다. 따라서 *TreeFill*알고리즘이 사용하는 라운드의 수는 다음과 같다.

$$\lceil \log(w/n) \rceil + 2 = O(\log(w/n)) \tag{3.1}$$

이제 각 라운드에서 *DetectTree*가 n개의 코인을 검출하기 위해 사용하는 메시지의 수를 생각해 보자. *DetectTree*에서 n개 코인의 분산 검출을 위해 노드들 사이에 주고 받는 메시지는 $FULL$ 메시지와 코인이다. 노드들 사이에 주고 받는 $FULL$ 메시지의 수를 $NumFull$이라 하고 노드들 사이에 주고 받는 코인의 수를 $NumCoin$이라 하자. 그러면 각 라운드에서 *DetectTree*가 코인의 분산 검출을 위해 사용하는 메시지의 총 수는 $NumFull + NumCoin$이다.

$NumFull$은 쉽게 구할 수 있다. 루트 노드를 제외한 다른 노드들은 *DetectTree*의 알고리즘에 따라 각 라운드에서 하나의 $FULL$ 메시지를 부모에게 보낸다.

$$NumFull = (n-1) - 1 = n - 2. \tag{3.2}$$

이제부터 $NumCoin$이 높은 확률로 $O(n)$임을 보인다. 증명을 위해 다음과 같은 정의들이 필요하다.

정의 2 *X_i는 $0 < i \leq h$ 일 때, DetectTree의 i번째 계층에 속해 있는 어떤 하나의 노드에서 $i-1$번째 계층으로 전달된 코인의 총 수를 나타내는 랜덤 변수 이다.*

정의 3 *Y는 n개의 코인이 DetectTree에 도착할 때, 어떤 하나의 리프 노드가 받은 코인의 수를 나타내는 랜덤 변수 이다.*

정의 4 *R_i는 $0 \leq i < h$ 일 때, DetectTree의 i번째 계층에 있는 모든 노드가 $(i+1)$번째 계층에 있는 노드들로 부터 전달 받은 코인의 총 수를 나타내는 랜덤 변수이다.*

*DetectTree*의 i번째 계층에 있는 노드으 총 수는 2^i 이므로, R_{i-1}과 X_i는 다음과 같은 관계를 가진다.

$$R_{i-1} = \sum_{i=1}^{2^i} X_i = 2^i X_i. \tag{3.3}$$

랜덤 변수 X_h는 *DetectTree*의 한 리프 노드가 받은 코인의 수 에서 2를 뺀 값이다. 즉, 각 리프 노드는 X_h만큼의 코인을 상위 계층으로 전달하게 된다. 따라서 X_h와 Y는 다음과 같은 관계를 가진다.

$$X_h = Y - 2. \tag{3.4}$$

*DetectTree*의 알고리즘에 의해 어떤 한 코인이 $(i+1)$번째 계층에서 i번째 계층으로 전달 되었다면, 2개 미만의 코인을 가지고 있는 리프 노드에게 코인을 전달하기 위해 i번째 계층에서 $(i+1)$번째 계층으로 다시 전달 되어야만 한다. 이러한 관찰에서부터 $NumCoin$을 R_i로 부터 다음과 같이 유도할 수 있다.

$$NumFwd = 2 \cdot (R_{h-1} + R_{h-2} + \cdots + R_0) = 2 \sum_{i=0}^{h-1} R_i. \tag{3.5}$$

따라서 R_i에 관한 식을 구하면, $NumCoin$에 관한 식 역시 식 3.5로 부터 얻을 수 있다.

랜덤 변수 Y는 이항 분포 (binomial distribution) $\mathcal{B}(n, 2/n)$를 따른다. *TreeFill*의 각 라운드에서 최소한 n개의 코인이 *DetectTree*의 리프 노드들에게 무작위로 보

내진다. *DetectTree*의 리프 노드들의 수는 $2^h = 2^{h+1}/2 = n/2$이므로, 하나의 리프 노드가 코인을 받을 확률은 $1/(n/2) = 2/n$ 이다. 따라서 Y는 이항 분포 $\mathcal{B}(n, 2/n)$를 따른다. 랜덤 변수 Y의 기대값과 분산은 각각 $E[Y] = n(2/n) = 2$, $Var[Y] = n(2/n)(1-2/n) = 2(1-2/n)$ 이다.

이제 R_{h-1}를 식 3.3과 식 3.4를 사용하여 다음과 같이 Y를 이용하여 나타낼 수 있다.

$$R_{h-1} = 2^h X_h = 2^h(Y-2) = 2^h Y - n. \tag{3.6}$$

R_{i-1}과 R_i의 관계도 유도할 수 있는데 이를 위해 다음 정의가 필요하다.

정의 5 *m_d를 d번째 계층에 있는 내부 노드라 하자. SubTree(m_d)는 루트 노드가 m_d인 서브 트리로 정의 한다. NumRcv(m_d)는 n개의 코인이* DetectTree*에 도착하는 동안 SubTree(m_d)가 받은 코인의 총 수로 정의 한다.*

*SubTree(m_d)*에 있는 리프 노드의 총 수는 $2^h/2^d = 2^{h-d}$ 이다. 따라서 *NumRcv(m_d)*는 Y를 사용하여 다음과 같이 나타낼 수 있다.

$$NumRcv(m_d) = 2^{h-d}Y.$$

그러면 다음과 같은 정리가 성립한다.

정리 2 *TreeFill의 한 라운드에서 d번째 계층에 있는 한 내부노드 m_d는 $NumRcv(m_d) \geq 2 \cdot 2^{h-d}$인 경우 코인을 위쪽 계층인 $d-1$번째 계층으로 전달한다. n개 코인이 DetectTree에 도착하는 경우 이 조건을 만족하고 코인을 위쪽 계층으로 전달할 확률은 1/2이다.*

증명 *DetectTree*의 코인 분산 검출 알고리즘에 의해 m_d는 *SubTree(m_d)*의 모든 리프 노드들이 2개 씩의 코인을 받을 때 까지 $(d-1)$ 번째 계층으로 코인을 전달하지 않는다. 즉, $NumRcv(m_d) < 2 \cdot 2^{h-d}$인 경우 코인을 위쪽 계층으로 전달하지 않는다. 이 경우에는 2개 미만의 코인을 받은 리프 노드가 *SubTree(m_d)* 안에 있기 때문에 그 리프 노드에게 코인을 전달하면 된다. $Pr(NumRcv(m_d) \geq 2 \cdot 2^{h-d})$는 다음과 같다.

$$\begin{aligned} Pr(NumRcv(m_d) \geq 2 \cdot 2^{h-d}) &= Pr(2^{h-d}Y \geq 2 \cdot 2^{h-d}) \\ &= Pr(Y \geq 2). \end{aligned}$$

Y는 이항 분포 $\mathcal{B}(n, 2/n)$을 따른다. 충분한 수의 노드가 참여하는 경우 $\mathcal{B}(n, 2/n)$는 정규 분포 $\mathcal{N}(2, 2(1-2/n))$에 가까워지게 된다. 따라서 $Pr(Y \geq 2)$는 $Pr(\mathcal{N}(2, 2(1-2/n)) \geq 2)$로 나타낼 수 있으며 정규 분포의 특서에 의해 이 경우의 확률은 $1/2$이다.

정리 2에 의해 *DetectTree*의 내부 노드는 자신이 받은 코인들을 $1/2$의 확률로 위쪽 계층으로 전달한다. R_i는 i번째 계층의 노드들이 $(i+1)$번째 계층의 노드들로부터 받은 코인들의 수이다. 따라서 다음 관계가 성립한다.

$$R_i = \frac{1}{2}R_{i+1}, (0 \leq i < h). \tag{3.7}$$

이제 식 3.5, 3.6, 3.7을 사용하여 다음과 같이 $NumCoin$을 나타낼 수 있다.

$$\begin{aligned} NumCoin &= 2\sum_{i=0}^{h-1} R_i = 2\sum_{i=0}^{h-1} 2^{-i} R_{h-1} \\ &= 2(2^h Y - n) \cdot \frac{1-2^{-h}}{1-2^{-1}} \\ &= 4(1-2^{-h})(2^h Y - n) \\ &= 2n(1-2^{-h})Y - 4n(1-2^{-h}) \\ &= 2n(1-2^{-h})Y - E[2n(1-2^{-h})Y]. \end{aligned}$$

위 식에서 $\mu = E[2n(1-2^{-h})Y] = 4n(1-2^{-h})$이라 하자. 그러면 체르노프 부등식을 (Chernoff inequality) 사용하여 $\delta \leq 2e-1$를 만족하는 수 δ에 대해 $Pr(NumCoin > \delta\mu)$을 다음과 같이 구할 수 있다.

$$\begin{aligned} Pr(NumCoin > \delta\mu) &= Pr(2n(1-2^{-h})Y - \mu > \delta\mu) \\ &= Pr(2n(1-2^{-h})Y > (1+\delta)\mu) \\ &< e^{-\mu\delta^2/4}. \end{aligned}$$

따라서 다음 식이 성립한다.

$$1 - Pr(NumCoin > \delta\mu) = Pr(NumCoin \leq \delta\mu) > 1 - e^{-\mu\delta^2/4}.$$

$\mu = 4n(1-2^{-h}) \leq 4n$을 만족 하므로, $Pr(NumCoin \leq \delta\mu) < Pr(NumCoin \leq 4\delta n)$이 성립한다. δ를 1로 넣으면 $Pr(NumCoin \leq 4n)$의 하한을 (lower bound) 다음과

같이 얻을 수 있다.

$$\begin{aligned} Pr(NumCoin \leq 4n) &> Pr(NumCoin \leq \mu) \\ &> 1 - e^{-\mu/4} \\ &= 1 - e^{-n+2}. \end{aligned}$$

그러므로 $Pr(NumCoin \leq 4n) > 1 - e^{-n+2}$이다. 즉, $NumCoin$은 $1 - e^{-n+2}$보다 큰 확률로 $O(n)$이 된다.

식 에서 보인 바와 같이 *TreeFill*이 사용하는 라운드의 수는 $O(\log(w/n))$이다. 각 라운드에서 *DetectTree*가 사용하는 메시지의 수는 $NumFull + NumCoin$인데, $NumCoin$은 식 에서 보인 바와 같이 $O(n)$이다. 따라서 *TreeFill*알고리즘의 메시지 복잡도는 다음과 같다.

$$O(\log(w/n)) \cdot (NumFull + NumCoin) = O(n \log(w/n)).$$

$NumCoin$이 $O(n)$의 확률은 $1-e^{-n+2}$보다 크므로 *TreeFill*은 높은 확률로 $O(n \log(w/n))$의 메시지 복잡도를 만족 한다.

3.3.2 *TreeFill*의 MaxRcv

이 절에서는 *TreeFill*의 각 노드에서 받은 메시지 수의 최대 값, MaxRcv가 높은 확률로 $O(\log(w/n))$ 임을 증명한다.

*DetectTree*의 노드가 받는 코인의 수는 부모 노드가 전달해 주는 코인과 아래쪽 계층에서 전달되는 코인의 수의 합이다. 이렇게 위쪽, 아래쪽에서 전달되는 코인의 수를 다음과 같이 정의한다.

- $RcvUp_i$: *TreeFill*의 각 라운드에서, i번째 계층에 있는 노드가 부모로부터 전달 받는 코인의 수.
- $RcvDn_i$: *TreeFill*의 각 라운드에서, i번째 계층에 있는 노드가 $(i+1)$번째 계층에 있는 노드들로부터 전달 받는 코인의 수.

*DetectTree*의 i번째 계층에 있는 노드가 받는 메시지의 수를 $NumRcv_i$라 하자. 그러면 $NumRcv_i = RcvUp_i + RcvDn_i$ 이다. $NumRcv^{Rnd}$를 *TreeFill*의 한 라운드에서 각 노드가 수신하는 메시지의 최대값이라 하자. 그러면 $NumRcv_i$로 부터 $NumRcv^{Rnd}$를 다음과 같이 나타낼 수 있다.

$$MaxRcv^{Rnd} = max_{0 \le i \le h}(NumRcv_i).$$

*TreeFill*에게 필요한 라운드의 수는 $O(\log(w/n))$이므로 MaxRcv는 다음과 같이 나타낼 수 있다.

$$\text{MaxRcv} \le O(\log(w/n)) \cdot MaxRcv^{Rnd}.$$

이제 $NumRcv^{Rnd}$가 높은 확률로 $O(1)$임을 보임으로써 MaxRcv가 높은 확률로 $O(\log(w/n))$ 임을 보이고자 한다.

정의 에서 R_i는 i번째 계층의 노드들이 $(i+1)$번째 계층의 노드들에게서 받은 코인의 총 수임을 정의 하였다. 따라서 i번째 계층에 있는 노드의 수는 2^i이므로 $RcvDn_i = R_i/2^i$으로 나타낼 수 있다. 랜덤 변수 Y는 *DetectTree*의 한 리프 노드가 받는 코인의 수를 나타낸다. 랜덤 변수 Y와 식 3.6, 3.7에서 $RcvDn_i$는 다음과 같이

나타낼 수 있다.

$$\begin{aligned} RcvDn_i = R_i/2^i &= 2^{-(h-1-i)}R_{h-1}/2^i \\ &= 2^{-(h-1)}(2^hY - n) \\ &= 2(Y-2). \end{aligned}$$

이제 $RcvDn_i$를 Y를 이용하여 나타내었으므로, $RcvDn_i$의 통계 분포는 Y로 부터 유도할 수 있다. $RcvUp_i$에 대한 식을 구하면 $NumRcv_i$를 유도할 수 있다.

*DetectTree*의 내부 노드들은 정리 2에 의해 자신이 받은 코인들을 1/2의 확률로 위쪽 계층으로 전달하고, 역시 1/2의 확률로 자신의 자식들에게 전달한다. i번째 계층에 있는 내부 노드 m_i는 계층 $0 \sim (i-1)$에 존재하는 자신의 조상 노드들이 아래쪽으로 전달하는 코인을 받게 된다.

조상 노드들이 아래쪽으로 전달하는 코인들은 조상 노드의 좌측, 우측 서브 트리에 명백하게 같은 확률로 코인을 전달해야 한다. 왜냐 하면 서브 트리에 코인을 전달하는 확률은 각 서브 트리가 수신한 코인의 수와 관련이 있는데 모든 리프 노드가 같은 확률로 코인을 받으므로 통계적으로 특정 서브 트리가 더 많거나 적은 코인을 받는다고 할 수 없기 때문이다.

이러한 사실로 부터 $RcvUp_i$는 m_i의 조상 노드들로 부터 m_i에게 전달된 코인의 수 이므로 다음과 같이 나타낼 수 있다.

$$\begin{aligned} RcvUp_i &= (1/2)\cdot\{2^{-1}RcvDn_{i-1} + 2^{-2}RcvDn_{i-2} + \cdots + 2^{-i+1}RcvDn_1\} + 2^{-i}RcvDn_0 \\ &= (1/2)\sum_{j=1}^{i-1} 2^{-j}RcvDn_{i-j} + 2^{-i}RcvDn_0 \\ &= (Y-2)(1 - 2^{-i+1} + 2^{-i+1}) = (Y-2). \end{aligned}$$

위 식에서 $RcvDn_0$는 루트 노드가 받은 코인의 수이고, 루트 노드는 부모가 없기 때문에 자신이 받은 모든 코인이 i번째 계층의 내부 노드에게 같은 확률로 나누어 전달 된다. 반면, $1 \sim (i-1)$ 계층의 내부 노드들은 자신들이 받은 코인의 반은 위쪽 계층으로 전달하고 나머지 반을 자식들에게 전달하기 때문에 1/2이 곱해진 것이다.

따라서 $NumRcv_i$는 다음과 같이 나타낼 수 있다.

$$NumRcv_i = RcvUp_i + RcvDn_i = 3(Y-2).$$

$NumRcv_i$는 i와 관계 없이 $3(Y-2)$으로 나타난다. 따라서 $MaxRcv^{Rnd}$ 역시 $3(Y-2)$ 이다.

체르노프 부등식을 사용하면 $\delta \leq 2e-1$를 만독하는 δ에 대해 $Pr(MaxRcv^{Rnd} \leq 6\delta)$를 다음과 같이 얻을 수 있다.

$$\begin{aligned} Pr(MaxRcv^{Rnd} \leq 6\delta) &= 1 - Pr(MaxRcv^{Rnd} > 6\delta) \\ &\geq 1 - Pr(3(Y-2) > 6\delta) \\ &= 1 - Pr(Y-2 > 2\delta) \\ &= 1 - e^{-\delta^2/2}. \end{aligned}$$

따라서 $Pr(MaxRcv^{Rnd} \leq 6\delta)$는 $\delta < 2e-1$를 만족하는 δ에 대해 $1-e^{-\delta^2/2}$ 보다 크다. 예를 들어 $\delta = 2$인 경우 $MaxRcv^{Rnd}$가 12보다 작을 확률은 0.86보다 크다. 만약 $\delta = 3$ 이라면 $MaxRcv^{Rnd}$가 18보다 작을 확률은 0.99보다 커지게 된다.

MaxRcv는 $MaxRcv^{Rnd}$를 사용하여 다음과 같이 나타낼 수 있음을 앞에서 보였다.

$$\text{MaxRcv} \leq O(\log(w/n)) \cdot MaxRcv^{Rnd}.$$

$MaxRcv^{Rnd}$가 높은 확률로 $O(1)$이므로 MaxRcv는 높은 확률로 $O(\log(w/n))$이 된다.

제 4 장 분산 트리거 계수를 위한 효율적인 확률적 알고리즘 (Probabilistic Algorithm)

본 장에서는 분산 트리거 계수 문제를 보다 효율적으로 해결하는 확률적 알고리즘인 *TreeFill-p*를 제시 한다. 3의 *TreeFill*은 정확한 알고리즘 (exact algorithm) 이다. n개의 노드에 w개의 메시지가 도착한 경우 메시지 손실이 없다면 반드시 사용자에게 이 사실을 알려준다. Garg가 증명한 분산 트리거 계수 문제를 해결하는 알고리즘들의 메시지 복잡도의 하한은 실패 확률이 없는 정확한 분산 트리거 계수 알고리즘들을 대상으로 하는 것이다 [GGS10].

*TreeFill-p*는 확률적 알고리즘으로써 낮은 실패 확률을 가지고 있다. 즉, w개의 트리거가 n개의 노드에 도착하고 메시지 손실이 없어도 낮은 확률로 사용자에게 이 사실을 알리지 못 할 수 있다. 그러나 확률적 알고리즘인 *TreeFill-p*는 *TreeFill*과 비교하여 더욱 우수한 메시지 복잡도를 가진다. *TreeFill-p*는 높은 확률로 $O(n)$의 콘트롤 메시지들을 사용하여 w개의 트리거를 검출해 낸다. 트리거 검출을 위한 메시지 오버 헤드는 전체 노드에 고르게 분산 된다. 따라서 *TreeFill-p*의 MaxRcv는 $O(1)$이 된다.

4.1절에서 *TreeFill-p* 알고리즘에 대해 설명한다. 5.5절에서 *TreeFill-p* 알고리즘에 필요한 라운드의 수를 분석한다. 다음으로 4.3절에서 *TreeFill-p* 알고리즘의 성공 확률을 분석한다. 마지막으로 4.4에서 *TreeFill-p*알고리즘의 성능 분석을 보인다.

4.1 *TreeFill-p* 알고리즘

3장의 *TreeFill* 알고리즘과 *TreeFill-p* 알고리즘은 같은 시스템 모델을 사용한다. 또한 *TreeFill-p* 알고리즘 역시 n개의 코인을 검출하기 위해 *TreeFill* 알고리즘에서 사용하던 *DetectTree*를 사용한다. 또한 *TreeFill-p* 역시 라운드 기반으로 작동 한다. 하지만 *TreeFill-p* 알고리즘은 확률적인 방식으로 코인을 생성한다. *TreeFill-p* 알고리즘에서도 설명의 단순성을 위하여 전체 노드 수 n은 어떤 정수 h에 대하여 $n = 2^{h+1}$이라고 가정한다.

TreeFill 알고리즘은 w개의 트리거가 n개의 노드에 도착할 때 최소한 n개의 코인이 발생함을 이용 하였다. n개의 코인이 반드시 발생하기 때문에 이 코인들을 *DetectTree*를 이용하여 분산 검출하면 전체 트리거들의 발생을 분산 검출할 수 있었다.

*TreeFill-p*알고리즘에서 각 노드는 트리거를 받을 때 마다 일정 확률로 코인을 발생 시킨다. 첫번째 라운드에서 각 노드는 트리거가 도착할 때 마다 n/w의 확률로 코인을 발생 시킨다. 그리고 발생한 코인은 *TreeFill* 알고리즘에서와 같이 $n-1$개의 노드로 구성된 *DetectTree*의 리프 노드 중 하나를 무작위로 선택하여 보낸다. 만약 발생한 코인의 수가 n보다 크거나 같으면 *DetectTree*는 n개의 코인이 도착한 순간을 감지할 수 있다.

TreeFill 알고리즘 에서와 같이 *TreeFill-p* 알고리즘에서도 w_i를 i번째 라운드가 시작할 때 아직 검출되지 않은 트리거의 수라고 정의한다. 그러면 i번째 라운드에서 각 노드가 코인을 발생시키는 확률은 n/w_i 이다.

랜덤 변수 D_i를 i번째 라운드에서 발생하는 코인의 수라고 하자. 첫번째 라운드에서 발생하는 코인 수의 기대값은 $E[D_1] = w \cdot (n/w) = n$ 이다.

각 노드에서 코인을 발생시키는 사건은 서로 독립이며 각 라운드에서 n/w_i로 일정하다. 따라서 D_i의 분포는 이항 분포 $\mathcal{B}(w_i, n/w_i)$를 따른다. 충분히 큰 n과 w에 대해 D_i는 정규 분포 $\mathcal{N}(n, n(1-n/w))$로 나타낼 수 있다.

각 라운드가 시작할 때 $c > 0$인 상수 c에 대하여 $c\sqrt{n}$개의 코인을 *DetectTree*의 리프 노드들에게 배포한다. 그러면 i번째 라운드에서 w_i개의 트리거가 도착할 때 노드들이 생성한 코인이 $n-c\sqrt{n}$ 이상이면 *DetectTree*의 리프 노드들이 가지고 있는 코인의 수는 $c\sqrt{n}+(n-c\sqrt{n}) = n$이 된다. 따라서 노드들에 의해 $n-c\sqrt{n}$ 개 이상의 코인이 발생하면 *DetectTree*가 이를 감지할 수 있다. 적절한 상수 c를 선택함으로써 i번째 라운드에서 노드들이 $n-c\sqrt{n}$개 이상의 코인을 생성할 확률을 충분히 높게 할 수 있다. c 값에 따른 각 라운드의 성공 확률 및 *TreeFill-p* 알고리즘의 성공 확률은 4.3절에서 살펴본다.

*DetectTree*의 루트 노드가 *FULL* 메시지를 받으면 *DetectTree*의 리프 노드들이 n개의 코인을 가지고 있는 것이다. 이 때 루트 노드는 *TreeFill* 알고리즘과는 달리 정확하게 몇 개의 트리거들이 n개의 노드들에 의해 해당 라운드에서 감지 되었는지 알 수 없다. 따라서 전체 노드가 받은 정확한 수의 트리거들을 취합 (aggregation) 해야 한다. 이를 위해 *DetectTree*를 이용한다. i번째 라운드에 노드들에 의해 검출된 트리거들의 수를 취합 하려면 $O(n)$개의 메시지가 필요 하다.

TreeFill-p 알고리즘의 라운드가 진행 되면서 i번째 라운드에 남아 있는 트리거의 수가 $w_i \leq n$이 되면 각 노드는 트리거를 검출할 때 마다 코인을 만든다. 만약 i번째 라운드가 시작할 때 $w_i < n$이라면 노드들에 의해 생성되는 코인의 수가 n보다 작으므로 미리 $n - w_i$개의 코인을 *DetectTree*의 리프 노드들에게 배포한다. 이를 통해 *DetectTree*는 남아 있는 w_i개의 코인들이 도착하는 시점을 알 수 있다.

알고리즘 4.1는 *TreeFill-p*의 알고리즘을 보여준다.

알고리즘 4.1 *TreeFill-p* 알고리즘

i번째 라운드가 시작 할 때:
  만약 $n/\hat{w}_i < 1$ 이라면:
    $c\sqrt{n}$개의 코인을 *DetectTree*의 리프 노드들에게 배포.
  그렇지 않다면:
    $(n - w_i)$개의 코인을 *DetectTree*의 리프 노드들에게 배포.

각 노드 m이 트리거를 받았을 때:
  만약 $n/\hat{w}_i < 1$ 이라면:
    $n/\hat{w}_i$의 확률로 코인을 리프 노드 중 하나에게 무작위로 전송.
  그렇지 않다면:
    코인을 리프 노드 중 하나에게 무작위로 전송.

루트 노드가 *FULL* 메시지를 받았을 때:
  i번째 라운드에서 노드들이 받은 트리거의 수를 *DetectTree*를 이용하여 취합.
  $w_{i+1} \leftarrow w_i -$ (취합된 트리거의 수).
  다음 라운드를 시작.

4.2 분산 트리거 계수를 위해 *TreeFill-p* 알고리즘이 사용하는 라운드 수

이 절에서는 *TreeFill-p* 알고리즘이 w개의 트리거를 검출하기 위해 필요한 라운드의 수를 분석한다.

i 번째 라운드가 시작할 때 $w_i > n$ 인 경우 $c > 0$인 상수 c에 대해 $c\sqrt{n}$개의 코인이 *DetectTree*의 리프 노드들에 배포된다. 각 노드는 트리거를 받을 때 마다 n/w_i의 확률로 코인을 발생 시키므로 $n - c\sqrt{n}$개의 코인이 노드들에게서 *DetectTree*에 전송되면 해당 라운드는 종료된다.

D_i는 i번째 라운드에서 노드들에 의해서 생성된 코인의 수를 나타내는 랜덤

변수라 하자. 그려면 $D_i \geq n - c\sqrt{n}$인 경우 i번째 라운드는 종료 가능 하다.

$Pr(D_i \geq n - c\sqrt{n})$는 체르노프 부등식을 이용하여 다음과 같이 구할 수 있다.

$$Pr(D_i \geq n - c\sqrt{n}) = 1 - Pr(D_i < n - c\sqrt{n}) \geq 1 - e^{-c^2/2}. \tag{4.1}$$

따라서 c값이 증가할수록 i번째 라운드가 성공적으로 종료할 확률 역시 증가한다.

이제 $D_i \geq n - c\sqrt{n}$를 만족하기 위해 w_i개의 트리거 중 노드들에 의해 감지되어야만 하는 트리거들의 비율은 어느 정도 인지 다음 정리를 통해 알아본다.

정리 3 *$D_i \geq n - c\sqrt{n}$를 만족 하려면 w_i개의 트리거 중 최소한 $w_i \cdot (1 - c/\sqrt{n})$개의 트리거들은 노드들에 의하여 감지 되어야 한다.*

증명 i번째 라운드가 시작할 때 w_i개의 트리거는 아직 검출되지 않은 상태 이다. i번째 라운드에서 검출되는 트리거의 비율을 α라 하자. 그러면 i번째 라운드에서 검출되는 트리거의 수는 αw_i 이며 α의 범위는 $0 \leq \alpha \leq 1$ 이다.

αw_i개의 트리거가 노드들에 의하여 검출될 때 발생하는 코인 수를 랜덤 변수 $\hat{D}_i$로 나타내기로 하자. 그러면 $\hat{D}_i$의 기대 값은 $E[\hat{D}_i] = \alpha w_i \cdot n/w_i = \alpha n$ 이다. 체르노프 부등식을 사용하여 $d > 0$인 상수 d에 대하여 다음과 같이 $Pr(\hat{D}_i \leq \alpha n - d\sqrt{\alpha n})$를 구할 수 있다.

$$\begin{aligned} Pr(\hat{D}_i \leq \alpha n - d\sqrt{\alpha n}) &= Pr(\hat{D}_i \leq (1 - d/\sqrt{\alpha n})\alpha n) \\ &< e^{-d^2/2}. \end{aligned}$$

따라서 $Pr(\hat{D}_i > \alpha n - d\sqrt{\alpha n}) \geq 1 - e^{-d^2/2}$이고, αw_i개의 트리거들이 노드들에 의하여 감지 될 때에 $1 - e^{-d^2/2}$ 보다 큰 확률로 $n - d\sqrt{\alpha n}$개의 코인이 발생함을 알 수

있다.

만약 $\hat{D}_i \geq n - c\sqrt{n}$ 라면 *DetectTree*는 $n - c\sqrt{n}$개 이상의 코인을 받을 수 있으므로 i번째 라운드는 종료 가능하다. 이제 $n - d\sqrt{\alpha n} \geq n - c\sqrt{n}$를 만족하기 위한 α의 범위를 다음과 같이 구해 보자.

$$\begin{aligned} \alpha n - d\sqrt{\alpha n} &\geq n - c\sqrt{n} \\ \sqrt{n}(\sqrt{\alpha})^2 - d\sqrt{\alpha} - \sqrt{n} + c &\geq 0 \\ \sqrt{\alpha} &\geq \frac{d + \sqrt{d^2 + 4\sqrt{n}(\sqrt{n} - c)}}{2\sqrt{n}} > \frac{d}{2\sqrt{n}} + \sqrt{\frac{n - c\sqrt{n}}{n}} \\ \alpha &> 1 + (d^2/4n) - c/\sqrt{n} \\ &> 1 - c/\sqrt{n}. \end{aligned} \tag{4.2}$$

따라서 충분히 큰 d 값을 선택하여 $\hat{D}_i$의 하한을 충분히 낮고 신뢰성 있게 설정하더라도 α는 $1 - c/\sqrt{n}$ 보다는 큼을 알 수 있다.

따름정리 1 *TreeFill-p알고리즘에서 i번째 라운드가 성공적으로 종료되면, $w_{i+1} \leq (c/\sqrt{n})w_i$ 이다.*

증명 정리 3에 의해 i번째 라운드가 성공적으로 종료될 때 노드들에 의하여 감지되는 트리거의 비율은 $(1 - c/\sqrt{n})$ 보다 크다. 따라서 w_{i+1}은 다음을 만족한다.

$$w_{i+1} \leq w_i - (1 - c/\sqrt{n})w_i = (c/\sqrt{n})w_i.$$

TreeFill-p 알고리즘은 $w_i \leq n$이 되면 각 노드가 감지하는 모든 트리거에 대하여 코인을 만들고, *DetectTree*는 n개의 코인을 감지할 수 있으므로, $w_i \leq n$을 만족하는 라운드가 마지막 라운드가 된다.

$w_1 = w$이고, 따름정리 1에 의하여 w_i는 다음과 같이 나타낼 수 있다.

$$w_i \leq (c/\sqrt{n})^{i-1} w.$$

따라서 어떤 정수 $o \geq 1$에 대하여 $(c/\sqrt{n})^{o-1} w \leq n$이면 $w_i \leq n$이 되고 *TreeFill-p*의 마지막 라운드가 수행된다. 이 때의 정수 o가 *TreeFill-p* 알고리즘이 필요로 하는 라운드 수의 상한이다.

w가 $O(n^m)$인 경우를 생각해 보자. 그러면 어떤 정수 c'에 대하여 위 식을 만족하는 $o = 2m + c'$를 찾을 수 있다. 따라서 w가 $O(n^m)$이면 *TreeFill-p*가 필요로 하는 라운드의 수는 $O(1)$이 된다.

4.3 *TreeFill-p* 알고리즘의 성공 확률

TreeFill-p 알고리즘이 성공하기 위해서는 w개의 트리거들이 n개 노드에 도착할 때에 모든 라운드에서 n개의 코인을 성공적으로 검출할 수 있어야 한다.

앞 절에서 보인 바와 같이 *TreeFill-p* 알고리즘은 $w = O(n^m)$ 일 때, $O(1)$ 라운드를 필요로 한다. 어떤 정수 o에 대해 $(c/\sqrt{n})^{o-1} w \leq n$를 만족하는 정수 o가 *TreeFill-p*가 필요로 하는 라운드의 수 이다. $w = n^m$인 경우 $(c/\sqrt{n})^{o-1} w \leq n$에서 $o \leq 2m - 1$를 얻을 수 있다.

식 4.1에서 보인 바와 같이, D_i가 i번째 라운드에서 노드들에 의해서 생성된 코인의 수를 나타내는 랜덤 변수라면 $D_i \geq n - c\sqrt{n}$인 경우 i번째 라운드는 종료 가능하며 그 확률은 $Pr(D_i \geq n - c\sqrt{n}) \geq 1 - e^{-c/2}$ 이다.

$w = n^m$인 경우 필요한 라운드 수는 최대 $2m - 1$ 이므로, 모든 라운드의 성공

확률을 곱한 *TreeFill-p*알고리즘의 성공 확률은 다음과 같다.

$$\prod_{i=1}^{2m-1} 1 - e^{-c/2} = (1 - e^{-c/2})^{2m-1}.$$

표 4.1는 $n = 1000$이고 w가 40000, 100000인 경우 c가 $2, 3, 4$일 때 *TreeFill-p*의 성공 확률을 정리한 것이다.

c	$w = 40000$	$w = 100000$
2	0.646	0.646
3	0.967	0.967
4	0.999	0.999

표 4.1: $n = 1000$이고 w가 $40000, 100000$인 경우 c 값에 따른 *TreeFill-p* 알고리즘의 성공 확률.

TreeFill-p 알고리즘의 성공 확률은 $w = n^m$인 경우 $(1-e^{-c/2})^{2m-1}$ 이다. 따라서 c가 증가할 수록 성공확률 역시 증가한다. 확률은 1보다 작은 값이므로 m이 커지면 성공 확률은 낮아진다. $m = \log_n w$이므로 w가 커지면 m이 증가하고 n이 커지면 m이 감소한다. 대규모 분산 시스템에서는 n이 큰 값이므로 m은 통상 작은 정수가 된다. 대부분의 경우, $c \geq 3$인 c를 택하면 *TreeFill-p*는 높은 확률로 성공할 수 있다.

4.4 *TreeFill-p* 알고리즘의 성능 분석

TreeFill-p 알고리즘의 각 라운드가 시작할 경우 $c\sqrt{n}$개의 코인을 배포한다. i번째 라운드에서 w_i개의 트리거가 도착하는 경우 전체 노드가 만들어 내는 코인 수의 기대 값은 n이다. *DetectTree*는 n개의 코인을 감지하기 위해 $O(n)$개의 메시지를 사용한다. 따라서 각 라운드에서 사용하는 메시지 수는 $O(n)$이다.

5.5절에서 살펴본 바와 같이 *TreeFill-p*가 사용하는 라운드의 수는 $w = O(n^m)$

인 경우 $O(1)$이다. 따라서 *TreeFill-p*의 메시지 복잡도는 $O(n)$이 된다.

*TreeFill-p*의 MaxRcv는 *DetectTree*의 MaxRcv에 라운드 수를 곱한 것이다. *DetectTree*에서 각 노드가 받는 메시지 수의 최대값은 $O(1)$이고, 라운드 수는 O(1)이므로 MaxRcv 역시 $O(1)$이 된다.

제 5 장 시뮬레이션 결과

이 장은 본 논문에서 제안한 *TreeFill* 및 *TreeFill-p* 알고리즘의 성능을 NetLogo [Net]를 이용하여 비교하기로 한다. NetLogo는 여러 과학기술 분야에서 널리 사용되는 에이전트 기반의 시뮬레이션 환경이다. NetLogo는 현재 가십 프로토콜 (gossip protocol) 및 기타 다양한 어플리케이션 케층의 네트워크 프로토콜, 소셜 네트워크, 적응적 복잡계 (complex adaptive systems), 프렉탈 이론, 사회 과학의 여러 주제들, 소셜 네트워크 등의 여러 연구 분야에 사용되고 있다.

NetLogo를 이용하여 *TreeFill* 및 *TreeFill-p*를 구현하고 두 알고리즘의 성능을 비교한다. 또한 기존의 분산 트리거 계수 알고리즘들 중에서 가장 좋은 성능을 보여주는 *CoinRand* 알고리즘 역시 NetLogo를 이용하여 본 논문에서 제안한 알고리즘들과 성능을 비교한다.

5.1절 에서는 NetLogo를 통하여 *TreeFill*, *TreeFill-p* 및 비교군인 *CoinRand*을 어떤 방식으로 구현했는지 설명한다. 5.2절에서는 알고리즘들의 메시지 복잡도를, 5.3절에서는 알고리즘들의 MaxRcv를 비교한다.

5.1 시뮬레이션 방식

NetLogo는 *틱* (tick)이라 불리는 논리적인 시간 단위에 따라 동작한다. NetLogo 시뮬레이션은 매번 새로운 틱이 시작할 때 마다 자동적으로 호출되는 'go' 프로시저 안에 시뮬레이션에 필요한 여러 다른 프로시저를 호출함으로써 다양한 시뮬레이션의 동작들이 일어나도록 한다. 통상 'go' 프로시저 마지막에서 'tick' 프로시저를 호출 하는데, 이 프로시저는 틱을 하나 증가 시키고 다음 번 틱이 시작되고 'go' 프로시저가 호출 되도록 한다.

틱은 논리적인 단위이다. 매 번 틱이 증가하고 'go' 프로시저가 호출되는데 이 함수에서 사용자가 원하는 모든 작업을 끝낸 다음, 'tick' 프로시저를 호출해야 다음 번 틱을 진행한다. 단일 쓰레드를 사용하여 프로시저들이 서로 호출하는 방식으로 시뮬레이션을 진행하므로 여러 에이전트들의 상호작용을 비교적 쉽게 모델링할 수 있다. 또한 시뮬레이션의 내부 상태를 쉽게 모니터링 할 수 있는 여러 기능들을 제공하고 있어서 다양한 분산 알고리즘의 핵심적 측면을 빠르게 구현하고 그 특성을 파악하는 시뮬레이션 도구로써 유용하다.

본 논문에서 제안한 알고리즘의 시뮬레이션을 위하여 매번 새로운 틱이 실행될 때 트리거를 생성한다. 그리고 이 트리거를 n개의 노드 중 하나에게 무작위로 보낸다. 그러면 *TreeFill*, *TreeFill-p* 및 비교군인 *CoinRand* 알고리즘의 정의에 따라 각 노드에서 코인을 만들고 이 코인들을 검출하는 방식으로 트리거를 감지한다. 동일한 n과 w에 대해 세 알고리즘들이 보이는 메시지 복잡도 및 MaxRcv를 비교하였다.

TreeFill 알고리즘의 *DetectTree*는 이진 트리를 기반으로 하지만 알고리즘을 쉽게 k진 트리로 (k-ary tree) 확장 가능하다. 5.2.2절에서 *DetectTree*를 k진 트리로 했을 경우 k 값에 따른 메시지 오버헤드의 변화를 살펴 보도록 한다.

그림 5.1은 $k = 3$, $n = 3^3$ and $w = 40000$일 경우 시뮬레이션의 모습을 보여

준다.

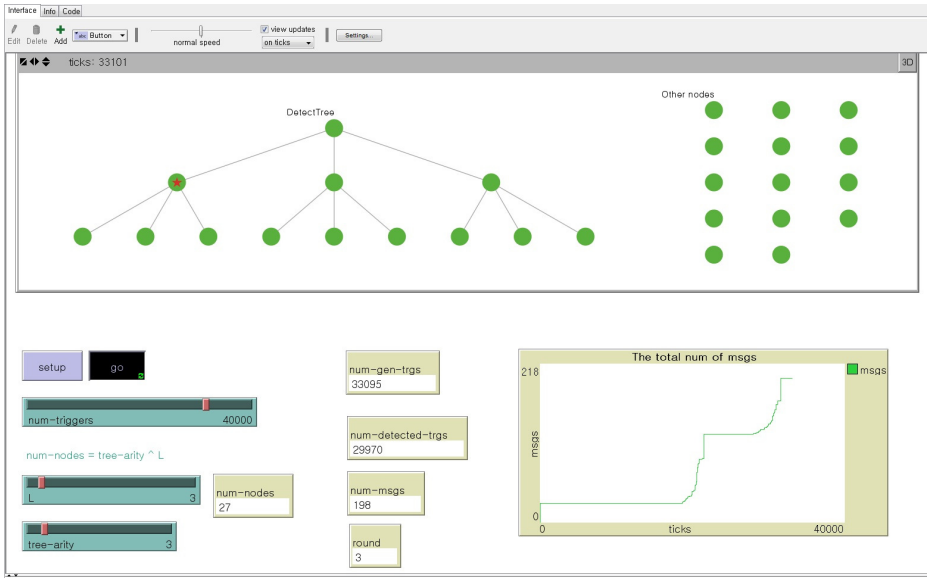

그림 5.1: $k = 3$, $n = 3^3$, $w = 40000$인 경우 NetLogo를 사용한 *TreeFill* 시뮬레이션.

5.2 메시지 복잡도 비교

5.2.1 노드 수 증가에 따른 메시지 복잡도 비교

CoinRand, *TreeFill*, *TreeFill-p* 알고리즘이 사용하는 메시지 수를 비교하기 위해 감지해야 하는 트리거의 수가 $w = 40000$인 경우, 전체 노드의 수를 $5 \leq i \leq 9$인 i에 대하여 $n = 2^i$로 놓고 모든 트리거를 분산 검출하기 위해 각 알고리즘이 사용한 메시지의 수를 그림 5.2와 같이 비교 하였다. 각 알고리즘이 사용한 메시지의 수는 모두 10회 반복 실험을 진행하고 평균을 구한 값이다.

$w = 40000$인 경우 *CoinRand*이 모든 경우 가장 많은 메시지를 사용하며 다른 두 알고리즘과의 차이는 노드 수가 증가할수록 커지고 있다. *CoinRand* 알고리즘의

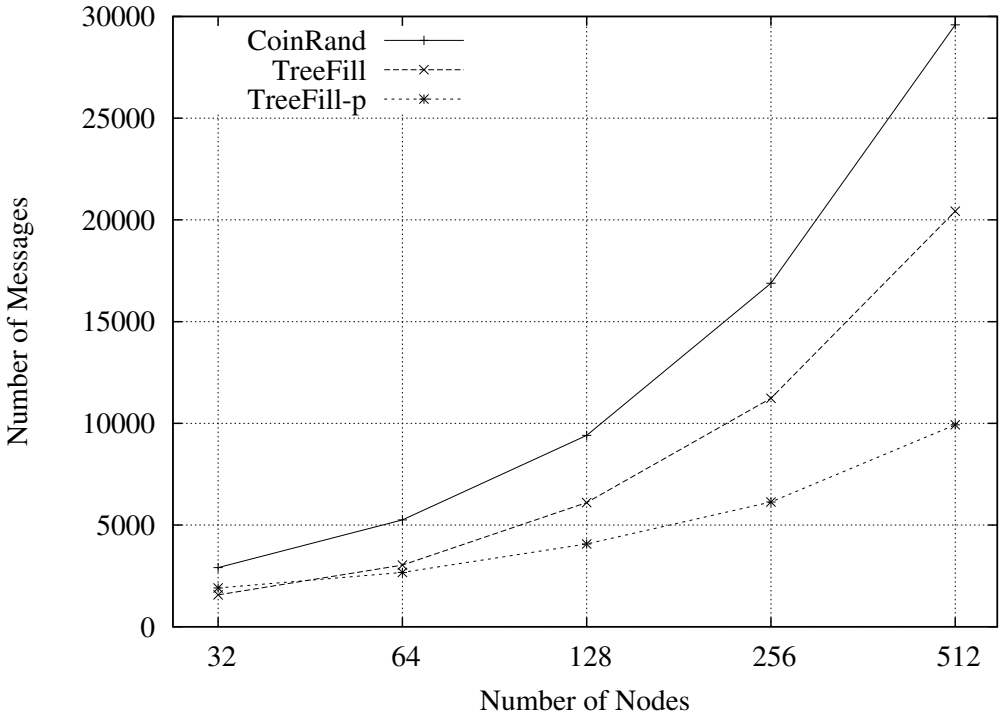

그림 5.2: 전체 노드 수 n이 $n = 2^i$ $(5 \leq i \leq 9)$인 경우 *CoinRand*, *TreeFill*, *TreeFill-p* 에서 사용한 전체 메시지 수.

메시지 복잡도는 $O(n(\log w + \log n))$ 이며, *TreeFill-p* 알고리즘의 메시지 복잡 도는 $O(n \log(w/n)) = O(n(\log w - \log n))$ 이다. $w \gg n$인 경우 두 알고리즘 모두 메시지 복잡도는 $O(n \log w)$이다. *CoinRand* 알고리즘도 최적에 가깝긴 하지만 최적의 메시지 복잡도를 가지고 있는 *TreeFill*에 비해 많은 메시지를 사용하고 있음을 확인할 수 있다. *TreeFill-p* 알고리즘은 다른 알고리즘에 비하여 훨씬 적은 메시지들을 분산 트리거 검출을 위해 사용하고 있다.

그림 5.3은 *TreeFill* 알고리즘의 시뮬레이션 결과가 알고리즘의 상한인 $O(n \log(w/n))$을 잘 따르는지 확인한 결과이다. 비교 결과 *TreeFill* 알고리즘은 $6.5n \log(w/n)$과 거의 일치하는 모습을 보여주고 있다.

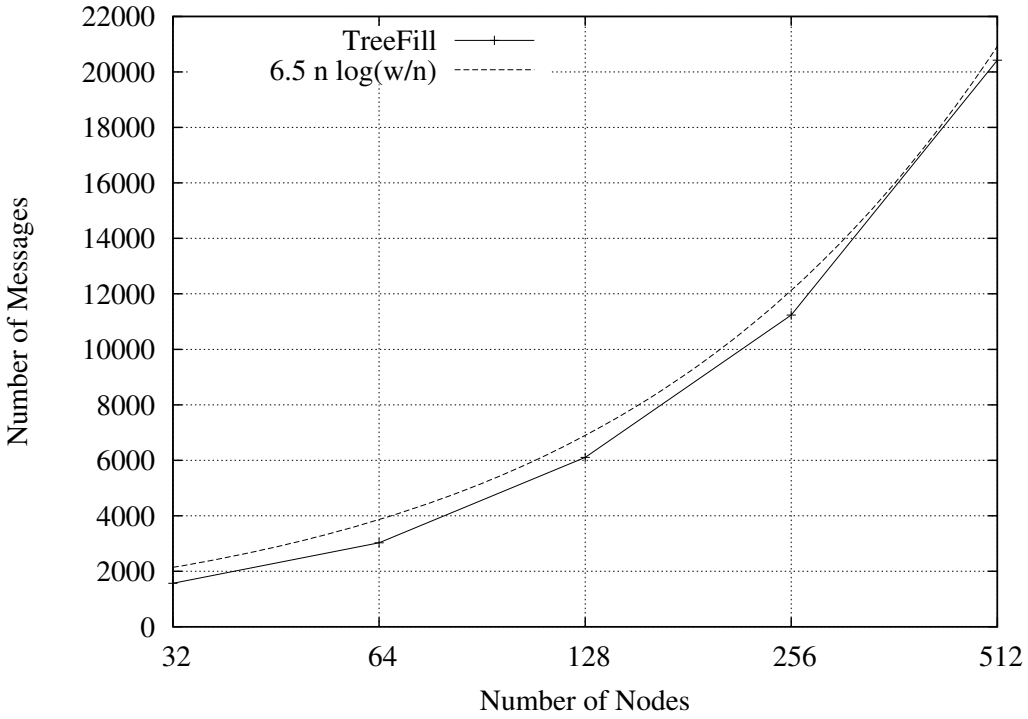

그림 5.3: *TreeFill* 알고리즘의 시뮬레이션 결과와 *TreeFill* 알고리즘 분석 결과 비교.

그림 5.4은 *TreeFill-p* 알고리즘의 시뮬레이션 결과가 알고리즘의 상한인 $O(n)$을 잘 따르는지 확인한 결과이다. 비교 결과를 보면 *TreeFill-p*는 n이 작을 때 예상보다 많은 메시지를 사용하고 있음을 알 수 있다.

이는 *TreeFill-p* 알고리즘의 각 라운드에서 남아있는 트리거를 검추하는 비율이 5.5절에서 보인 바와 같이 $1 - c/\sqrt{n}$ 보다 큰 값인데, $1 - c/\sqrt{n}$는 n이 작아짐에 따라 커진다. 표 5.1는 c가 $2, 3, 4$이고 n이 $32, 128, 512$인 경우 *TreeFill-p*의 각 라운드에서 검출 가능한 트리거의 비율을 보여준다.

표 5.1에서 보는 바와 같이 n이 증가함에 따라 각 라운드에서 더 많은 트리거를 검출하고, *TreeFill-p* 알고리즘은 n이 커질수록 더 적은 라운드를 필요로 한다. 그림 5.5는 그림 5.2의 시뮬레이션에서 *TreeFill-p* 알고리즘이 n에 따라 사용한 라

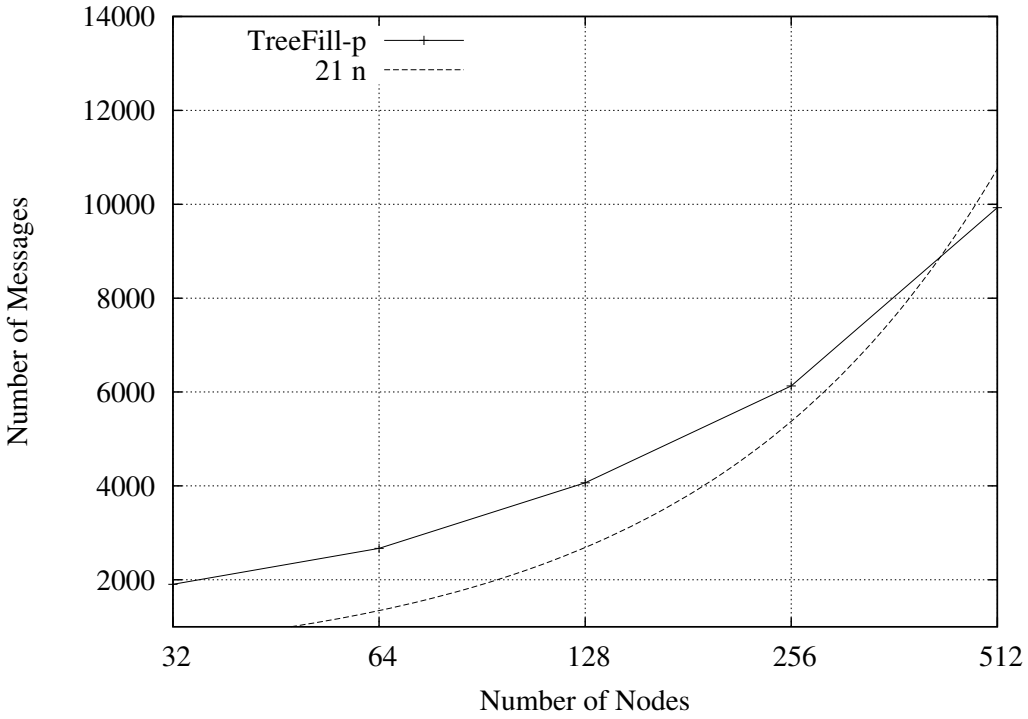

그림 5.4: *TreeFill-p* 알고리즘의 시뮬레이션 결과와 *TreeFill-p* 알고리즘 분석 결과 비교.

c	검출 비율($n = 32$)	검출 비율($n = 128$)	검출 비율($n = 512$)
2	0.65	0.82	0.91
3	0.47	0.73	0.87
4	0.29	0.65	0.82

표 5.1: c, n 값에 따라 *TreeFill-p* 알고리즘의 각 라운드에서 남아 있는 트리거를 검출하는 비율.

운드의 수를 보여준다.

TreeFill-p 알고리즘은 n이 작은 경우 많은 라운드를 사용하지만, 각 라운드에서 사용하는 메시지의 수는 *TreeFill-p* 알고리즘의 분석 결과에서 보인 바와 같이 $O(n)$의 성능을 보인다. 그림 5.6는 그림 5.2의 시뮬레이션에서 *TreeFill-p* 알고리즘이 각 라운드에서 사용한 메시지의 수를 보여준다.

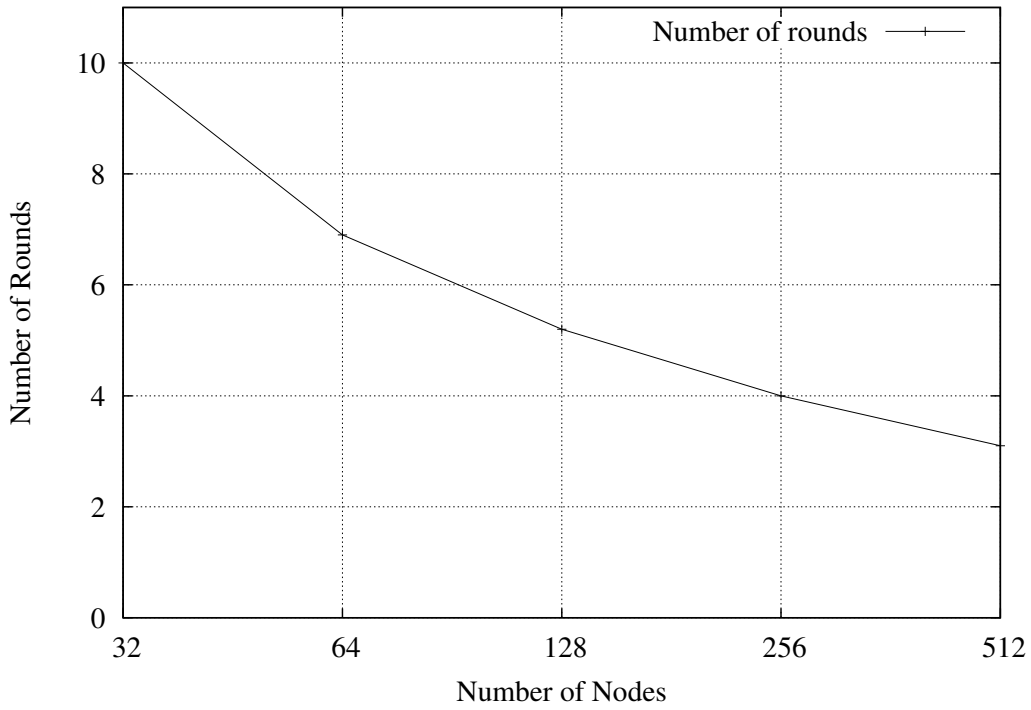

그림 5.5: *TreeFill-p* 시뮬레이션 결과에서 n에 따라 *TreeFill-p* 알고리즘이 사용한 라운드의 수.

5.2.2 k진 트리를 사용한 *TreeFill* 알고리즘

TreeFill 알고리즘은 이진 트리를 기반으로 설계되었지만 쉽게 k진 트리로 확장 가능하다. 이 절에서는 *TreeFill*알고리즘을 이진 트리에서 k진 트리로 확장한 실험 결과를 제시하고자 한다. k진 트리를 사용한 *TreeFill*알고리즘을 *TreeFill-k*라 하자. 그림 5.7는 $w = 40000$인 경우 *TreeFill*은 $n = 2^i$ $(5 \leq i \leq 9)$, *TreeFill-3*은 $n = 3^j$ $(3 \leq j \leq 6)$, *TreeFill-4*는 $n = 4^k$ $(2 \leq k \leq 5)$, *TreeFill-5*는 $n = 2^l$ $(2 \leq l \leq 4)$ 개의 노드를 사용한 경우 트리거 검출을 위해 사용한 메시지 수를 보여준다.

그림 5.7의 결과는 *TreeFill-5*이 같은 수의 트리거를 검출하기 위해 가장 적은 메시지를 사용했음을 보여준다. *TreeFill*의 경우 $6n\log(w/n)$과 유사한 수의 메시지를

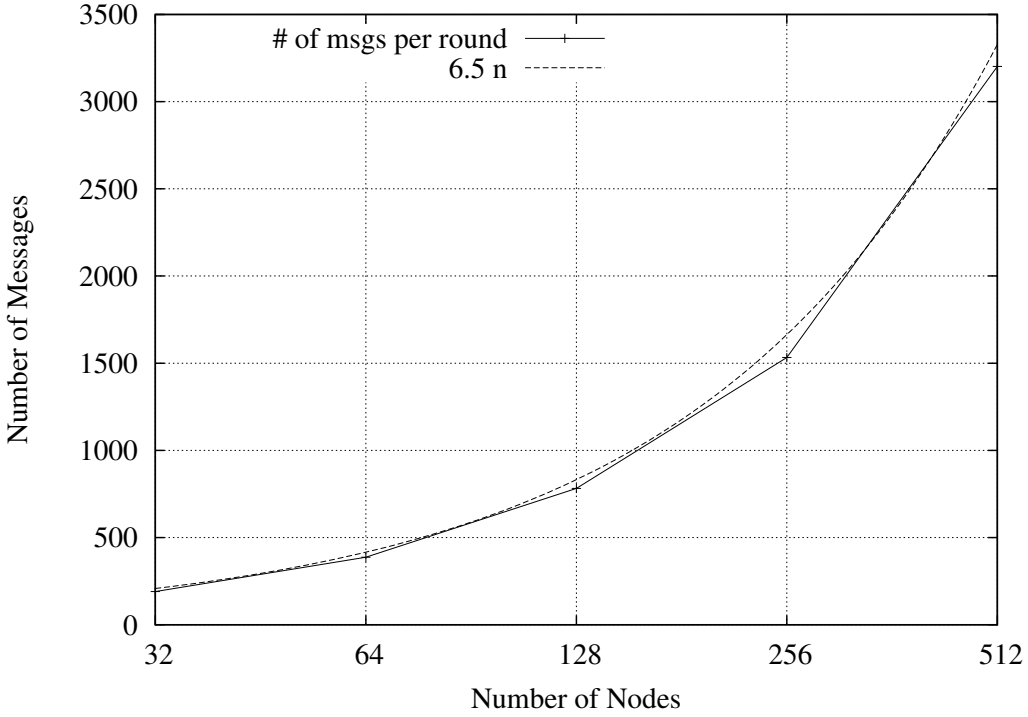

그림 5.6: *TreeFill-p* 시뮬레이션의 각 라운드에서 사용한 메시지의 수.

사용한데 비하여 *TreeFill-5*는 $3n\log(w/n)$과 비슷한 수의 메시지를 사용하였다. k진 트리를 이용한 *TreeFill-k*의 리프 노드들은 모두 n/k이며 k개의 코인을 받았을 때 $FULL$ 메시지를 보낸다. 그러면 루트 노드는 $FULL$ 메시지를 받았을 때 $k \cdot n/k = n$개의 코인이 도착했음을 확신할 수 있다. *TreeFill-k*에서는 *TreeFill*에 비해 더 적은 수의 노드가 코인 검출에 참여하며, $DetectTree$의 높이도 낮기 때문에 코인 검출을 위해 더 적은 수의 메시지를 사용한다. 따라서 *TreeFill-k*는 *TreeFill*에 비해 효율적이다.

그러나 더 적은 수의 노드가 코인 검출에 참여하므로 하나의 노드가 수신해야 하는 콘트롤 메시지의 수는 더욱 증가한다. 극단적으로 k를 계속 증가 시켜서 $n-1$이 되게 하면 *TreeFill-k*는 일종의 중앙 집중 방식의 알고리즘이 되고 루트 노드는

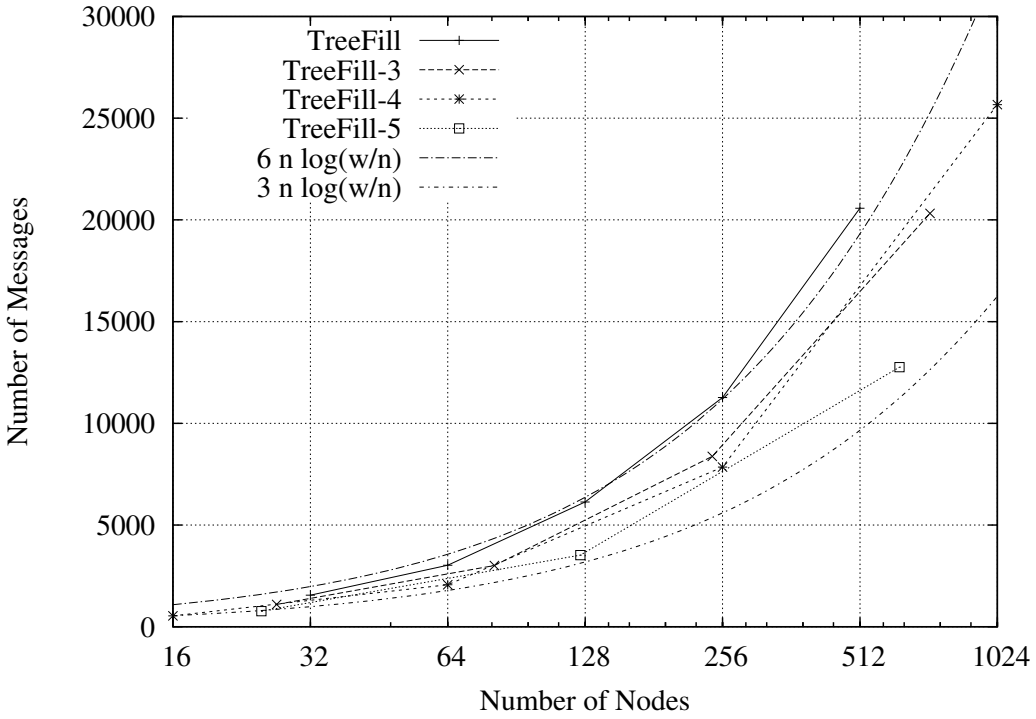

그림 5.7: 40000개의 트리거를 검출하기 위해 *TreeFill*, *TreeFill-3*, *TreeFill-4*, *TreeFill-5*가 사용한 전체 미시지 수.

n개의 코인을 검출하기 위해 $O(n)$개의 콘트롤 메시지를 처리해야 한다.

그렇다면 *TreeFill-k*에서 k 값의 증가에 따라 메시지 복잡도는 감소하지만 MaxRcv는 증가하게 되는데, 두 가지 성증 지표 상의 관계는 어떻게 될 것인지에 대한 문제가 제기된다. 이 부분에 대하여 추가적인 분석이 필요하며 이는 향후 연구 과제로 남겨둔다.

5.3 MaxRcv 비교

그림 5.8는 *CoinRand*, *TreeFill*, *TreeFill-p* 알고리즘의 MaxRcv를 비교한다. 실험에서 사용한 트리거의 수는 $w = 40000$이고 실험에 사용한 메시지의 수는 2^i $(5 \leq i \leq$

9) 이다. w와 n에 따른 각 실험은 10회 반복하였으며 그 평균값을 그림 5.8에 사용하였다. 그림 5.8에서의 수직 막대는 10회 반복 실험을 하는 도중 관찰된 MaxRcv의 최소 및 최대 값을 나타낸다.

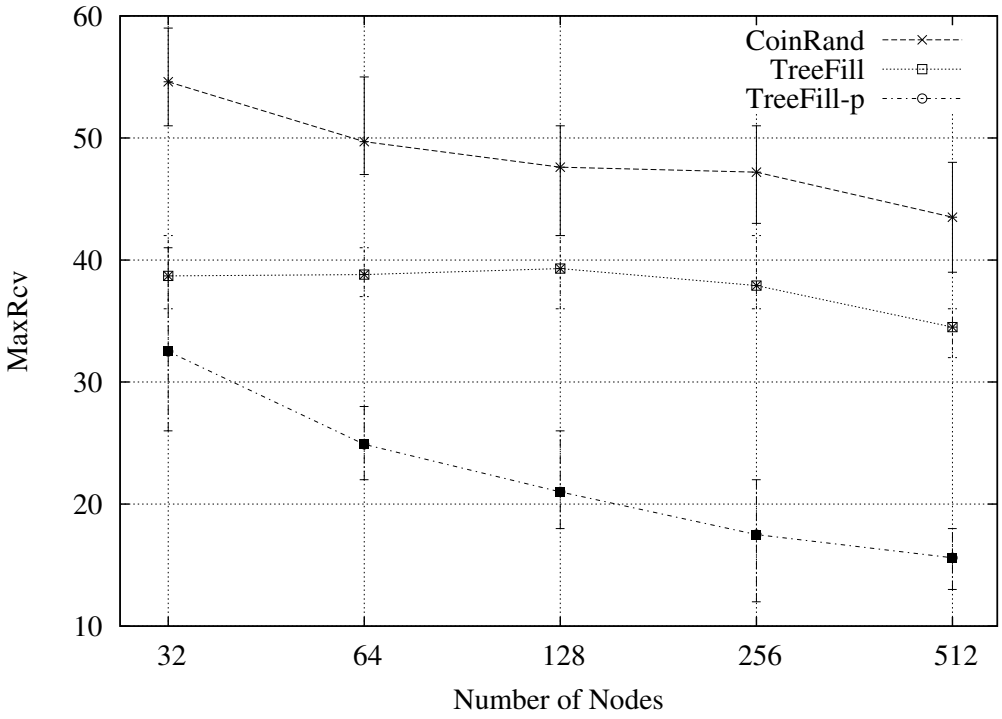

그림 5.8: *CoinRand*, *TreeFill*, *TreeFill-p* 알고리즘의 MaxRcv 비교.

CoinRand, *TreeFill*, *TreeFill-p* 알고리즘은 모두 라운드 기반으로 작동한다. 각 라운드에서 세 알고리즘의 MaxRcv 값은 모두 $O(1)$이다. 그런데 각 알고리즘이 필요로 하는 라운드의 수는 *CoinRand*이 $(O(\log n + \log w))$, *TreeFill*이 $(O(\log(w/n)))$, *TreeFill-p*이 $O(1)$이다. 그림 5.9는 세 알고리즘이 사용한 라운드의 수를 보여준다.

CoinRand 알고리즘이 가장 큰 MaxRcv를 보이는 것은 다른 알고리즘에 비해 많은 라운드를 사용하기 때문이다. *TreeFill-p*는 적은 수의 라운드만을 사용하므로 MaxRcv가 가장 작았다. 라운드 수에 비하여 MaxRcv의 증가가 가장 적은 알고리

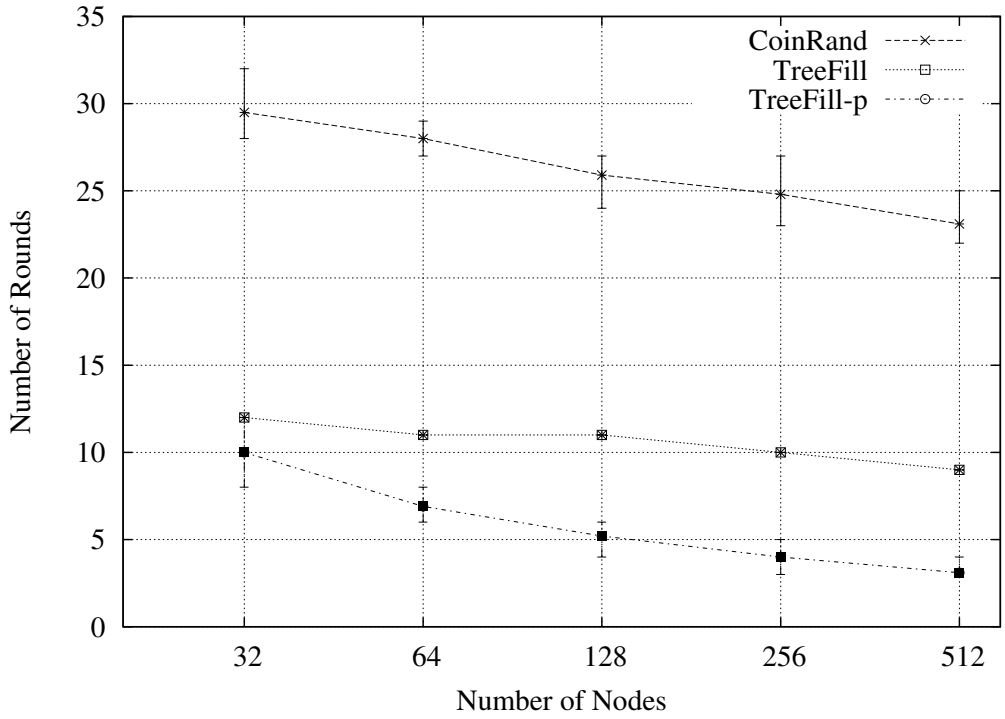

그림 5.9: 트리거들을 검출하기 위해 *CoinRand*, *TreeFill*, *TreeFill-p* 알고리즘이 사용한 라운드의 수.

즘은 *CoinRand*인데 이는 *CoinRand* 알고리즘의 각 라운드에서 각 노드가 수신하는 메시지의 수가 가장 작음을 의미한다. *TreeFill*이나 *TreeFill-p* 알고리즘은 모두 *DetectTree*를 사용하는데 여기서는 코인들이 트리의 위쪽으로 전달되었다가 아래쪽으로 내려오는 과정이 있다. 반면 *CoinRand* 알고리즘에서는 각각의 노드가 최대 2개의 코인을 위쪽으로 전달한다. 따라서 각 라운드에서 수신하는 메시지의 수는 *CoinRand*이 가장 작다. 그럼에도 불구하고 MaxRcv에서 라운드의 수가 더욱 지배적인 역할을 하고 있기 때문에 라운드 수가 적은 순서대로 MaxRcv 역시 우수한 결과를 보이고 있다.

제 6 장 결론

본 논문에서는 대규모 분산 시스템을 위한 효율적인 두 가지 분산 트리거 계수 알고리즘 *TreeFill*과 *TreeFill-p*를 제안 하였다.

분산 트리거 계수 알고리즘은 여러가지 분산 시스템의 주요한 기능 중 하나인 분산 모니터링을 위해 사용이 가능하다. 또한 분산 트리거 계수 알고리즘은 대규모 분산 시스템을 위한 효율적인 전역 스냅샷 알고리즘에 직접적으로 응용 된다. 따라서 효율적인 분산 트리거 계수 알고리즘은 대규모 분산 시스템의 분산 모니터링 및 전역 스냅샷을 위하여 유용하게 사용 가능하다.

분산 트리거 계수 문제는 시스템에 참여한 노드의 수가 n이고 검출해야 하는 트리거의 수가 w일 때 다음과 같이 정의된다.

- w개의 트리거를 n개의 노드에서 검출한다.
- 트리거들은 임의의 노드에 임의의 시간에 도착한다. 즉, 트리거의 도착 분포에 대한 어떠한 정보도 미리 주어지지 않는다.
- 전체 n개의 노드에서 검출한 트리거들의 총 합이 w가 되는 순간 사용자에게 이를 알려야 한다.
- $w \gg n$을 만족한다.

분산 트리거 계수 알고리즘의 성능 비교를 위한 지표로써 본 논문은 다음의 두 가지를 제시 하였다.

- 메시지 복잡도 (Message complexity): w개의 트리거를 검출하기 위해 n개의 노드들이 주고 받은 메시지 수의 총 합.
- 각 노드가 수신한 메시지 수의 최대 값 (MaxRcv): w개의 트리거를 검출하는 과정에서 가장 많은 메시지를 수신한 노드가 받은 메시지의 수.

TreeFill 알고리즘은 시스템 전체의 노드 수가 n이고 검출해야 하는 트리거의 수가 w라고 할 때 높은 확률로 $O(n\log(w/n))$의 메시지 복잡도를 보인다. 분산 트리거 검출을 위한 정확한 알고리즘들의 (exact algorithms) 메시지 복잡도의 하한은 $\Omega(n\log(w/n))$ 이다 [GGS10]. 따라서 *TreeFill* 알고리즘은 높은 확률로 최적의 메시지 복잡도를 보이는 정확한 분산 트리거 계수 알고리즘이다.

TreeFill 알고리즘이 트리거를 검출하기 위해 노드들 사이에서 주고받는 메시지들은 전체 노드에 고르게 분산된다. 따라서 *TreeFill*의 MaxRcv는 높은 확률로 $O(\log(w/n))$를 만족한다. 정확한 분산 트리거 계수 알고리즘들의 메시지 복잡도 하한은 $\Omega(n\log(w/n))$ 이므로 MaxRcv의 하한은 이러한 메시지 오버헤드가 전체 노드에 고르게 분산될 때 달성 가능하다. 따라서 MaxRcv의 하한은 $\Omega(\log(w/n))$ 이다. *TreeFill*의 MaxRcv는 높은 확률로 $O(\log(w/n))$ 이므로 *TreeFill*의 MaxRcv 역시 높은 확률로 최적의 성능을 보인다.

TreeFill-p 알고리즘은 확률적인 알고리즘이다 (probabilistic algorithm). w개의 트리거들을 n개의 노드로 검출할 때 *TreeFill-p* 알고리즘은 낮은 확률로 w개의 트리거가 모두 검출된 경우에도 사용자에게 이를 알리지 않을 수 있다. 하지만 이렇게 완화된 제한 조건을 대가로 *TreeFill-p* 알고리즘은 *TreeFill*알고리즘에 비하여 더 좋은 성능을 보인다.

어떤 정수 $m > 0$에 대하여 트리거의 수 $w = O(n^m)$이라 하자. 이 경우 *TreeFill-p*의 메시지 복잡도는 높은 확률로 $O(n)$이 된다. 또한 *TreeFill-p*역시 *TreeFill*과 마

찬가지로 메시지 오버헤드를 전체 노드에 고르게 분산 시킨다. 따라서 *TreeFill-p*의 MaxRcv는 $O(1)$이 된다.

TreeFill 및 *TreeFill-p*알고리즘을 통해 대규모 분산 시스템에서 분산 트리거 계수 문제를 효율적으로 해결할 수 있다. 이들 알고리즘은 다양한 분산 모니터링 문제와 대규모 분산 시스템의 전역 스냅샷 알고리즘을 위하여 유용하게 사용 가능하다.

참 고 문 헌

[ADBF+05] Sergio Andreozzi, Natascia De Bortoli, Sergio Fantinel, Antonia Ghiselli, Gian Luca Rubini, Gennaro Tortone, and Maria Cristina Vistoli. Gridice: a monitoring service for grid systems. *Future Generation Computer Systems*, 21(4):559–571, 2005.

[AK04] Ian F. Akyildiz and Ismail H. Kasimoglu. Wireless sensor and actor networks: research challenges. *Ad Hoc Networks*, 2(4):351–367, 2004.

[ASSC02] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[CCGS11] Venkatesan Chakaravarthy, Anamitra Choudhury, Vijay Garg, and Yogish Sabharwal. An efficient decentralized algorithm for the distributed trigger counting problem. *Distributed Computing and Networking*, pages 53–64, 2011.

[CCS11] Venkatesan T Chakaravarthy, Anamitra R Choudhury, and Yogish Sabharwal. Improved algorithms for the distributed trigger counting problem. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 515–523. IEEE, 2011.

[CDR08] Laukik Chitnis, Alin Dobra, and Sanjay Ranka. Aggregation methods for large-scale sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):9, 2008.

[CHL+06] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127. ACM, 2006.

[CL85] K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[CMY11] Graham Cormode, S. Muthukrishnan, and Ke Yi. Algorithms for distributed functional monitoring. *ACM Trans. Algorithms*, 7(2):1–20, 2011.

[EK10] Yuval Emek and Amos Korman. Efficient threshold detection in a distributed environment. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 183–191. ACM, 2010.

[GGS10] Rahul Garg, Vijay K Garg, and Yogish Sabharwal. Efficient algorithms for global snapshots in large distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5):620–630, 2010.

[JBA11] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *arXiv preprint arXiv:1110.0725*, 2011.

[KLPC13] Seokhyun Kim, Jaeheung Lee, Yongsu Park, and Yookun Cho. An optimal distributed trigger counting algorithm for large-scale networked systems. *Simulation: transactions of the society for modeling and simulation international*, 89(7):846–859, 2013.

[Ksh10] Ajay D Kshemkalyani. Fast and message-efficient global snapshot algorithms for large-scale distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(9):1281–1289, 2010.

[KXRE07] Ardalan Kangarlou, Dongyan Xu, Paul Ruth, and Patrick Eugster. Taking snapshots of virtual networked environments. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, page 4. ACM, 2007.

[LC10] Changlei Liu and Guohong Cao. Distributed monitoring and aggregation in wireless sensor networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[LY87] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4), 1993.

[MCC04] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[Net] NetLogo. http://ccl.northwestern.edu/netlogo/.

[ORS06] Adam J Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 14–23. ACM, 2006.

[PP06] KyoungSoo Park and Vivek S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74, 2006.

[SBF+04] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

[ss12] Top 500 supercomputer sites. http://www.top500.org, 2012.

[Tsa13] Jichiang Tsai. Flexible symmetrical global-snapshot algorithms for large-scale distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 24(3):493–505, 2013.

[ZC04] Wensheng Zhang and Guohong Cao. Dctc: dynamic convoy tree-based collaboration for target tracking in sensor networks. *Wireless Communications, IEEE Transactions on*, 3(5):1689–1701, 2004.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

Abstract

Let n be the number of nodes in a distributed system and w be the number of triggers to be detected with the nodes of the system. Distributed trigger counting algorithms raise an alert to a user, when the total number of triggers detected by the n nodes is reached to w. No statistical information on triggers is given in advance. The number of triggers is assumed to be much larger than the number of nodes.

Distributed trigger counting algorithms can be used for distributed monitoring and global snapshots. Distributed monitoring techniques are used to monitor the internal states or the surrounding environments of a system. Global snapshot algorithms are used to record entire system states as the check points of a system, so that when system failure occurs, a check point of the system can be used for system recovery.

In this thesis, we propose two efficient distributed trigger counting algorithms for large-scale distributed systems, *TreeFill* and *TreeFill-p*. Theses algorithms detect triggers based on rounds. In each round, a part of whole triggers is detected and if the number of detected triggers reaches to w, a user should be notified. In *TreeFill*, the total number of messages used to detect w triggers is $O(n \log(w/n))$ with high probability. This satisfies the lower-bound message complexity of the exact algorithms that solve the distributed trigger counting problem. While detecting the w triggers with the n nodes, the maximum number of messges received in each node

of the system is $O(\log(w/n))$ with high probability. *TreeFill-p* is a probabilistic algorithm; when n nodes have received w triggers, *TreeFill-p* may fail to detect the w triggers with low probability. However, when $w = O(n^m)$ for some constant $m > 0$, *TreeFill-p* detects w triggers using $O(n)$ messages with high probability. While detecting w triggers with *TreeFill-p*, the maximum number of messages received in each node is $O(1)$ with high probability. In this thesis, we prove the message complexities of *TreeFill* and *TreeFill-p*, and the maximum numbers of messages received in a node of *TreeFill* and *TreeFill-p*. Extensive simulations are also used to evaluate the performances of *TreeFill* and *TreeFill-p*.

Keywords: Distributed Trigger Counting Algorithms, Distributed Systems, Distributed Algorithms, Distributed Monitoring, Global Snapshots

Student Number: 2008-30213